EE 183DA: Design of Robotics System I

Lab Assignment 3

Group: Tiger

Loc Nguyen (104940808)

Huu Ngoc Tam Phan (704955811)

Man I Pang (305041386)

Tien Doan(104950425)

Date: February 19th, 2019

Abstract

For the previous labs, we mostly worked in continuous time and space for our robot to test for a computational system with different inputs and outputs. For this lab, we will work on discrete time and space to explore Markov Decision Processes (MDP) This process implements a mathematical framework for modeling decision making to control a simple discretized robot. The outcome of this process can be partly random and partly determined by the control of a decision maker. To understand MDP system in this lab, we will work with a simple robot which can be locate in any lattice point in a 2D grid world with length L and width W. For each part, there will be a task needed to complete in order to create a motion planning for the robot to reach the goal effectively from an initial starting point. Both policy iteration and value iteration are done to compare and analyze the results. Value iteration is this case shows more efficiency than policy iteration.

Part I. Introduction:

Markov Decision Process is a discrete time stochastic control process. At each time step, the process is in some state *s*, and the decision maker may choose any action *a* that is available in state *s*. The process responds at the next time step by randomly moving into a new state s', and giving the decision maker a corresponding reward $R_a(s, s')$.

A Markov decision process is a 4-tuple (S, A, P_a, R_a), where

- S is a finite set of states,
- A is a finite set of actions (alternatively, A_s is the finite set of actions available from state s),
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a

The core problem of MDPs is to find a "policy" for the decision maker: a function π that specifies the action π(s) that the decision maker will choose when in state *s*. The goal is to choose a policy π that will maximize some cumulative function of the random rewards, typically the expected discounted sum over a potentially infinite horizon:

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}) \quad \text{(where we choose } a_t = \pi(s_t)\text{, i.e. actions given by the policy)}$$

where γ is the discount factor and satisfies $0 \leq \gamma < 1$.

The standard family of algorithms to calculate this optimal policy requires storage for two arrays indexed by state: *value* V, which contains real values, and *policy* π which contains actions. At the

end of the algorithm, π will contain the solution and V(s) will contain the discounted sum of the rewards to be earned (on average) by following that solution from state s.

The algorithm has two kinds of steps, a value update and a policy update, which are repeated in some order for all the states until no further changes take place. Both recursively update a new estimation of the optimal policy and state value using an older estimation of those values.

$$\pi(s) := \arg\max_a \left\{ \sum_{s'} P(s'|s,a) \left(R(s'|s,a) + \gamma V(s') \right) \right\}$$

$$V(s) := \sum_{s'} P_{\pi(s)}(s,s') \left(R_{\pi(s)}(s,s') + \gamma V(s') \right)$$

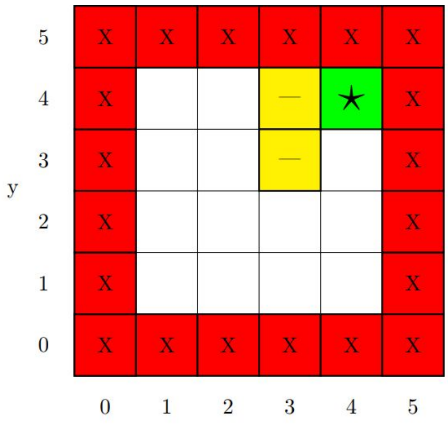
In value iteration which is also called backward induction, the π function is not used; instead, the value of π(s) is calculated within V(s) whenever it is needed. Substituting the calculation of π(s) into the calculation of V(s) gives the combined step

$$V_{i+1}(s) := \max_a \left\{ \sum_{s'} P_a(s,s') \left(R_a(s,s') + \gamma V_i(s') \right) \right\}$$

In policy iteration, step one is performed once, and then step two is repeated until it converges. Then step one is again performed once and so on.

This variant has the advantage that there is a definite stopping condition: when the array π does not change in the course of applying step 1 to all states, the algorithm is completed.

The state space we consider to analyze MDP is a 6 * 6 grid world (L = W = 6). The border states are in red, marked with an "X". Two barriers are placed at (3,3) and (3,4) in yellow, marked with "-". The goal cell are in green, marked with ⋆. The rewards for each state are independent of the actions taken by the robot. All border states have a reward of -100; barriers have a reward of -10'; and the goal square has a reward of 1. All other unmarked states has a reward of 0.



Part II. MDP System:

2.1(a). Create your state space S = {s}. What is the size of the state space N_S ?

Since we are considering the robot is in a 2D grid world of length L and width W where L=W=6 and twelve headings identified by the hours on a clock h ∈ {0 . . . 11}, the current state of the car can be defined as s=(x, y, h): 0 ≤ x < L, 0 ≤ y < W; x, y ∈ N and the state space N_S *is*

$$N_S = 6 \times 6 \times 12 = 432$$

```
S = []
for x in range(L):
    for y in range(L):
        for h in range(12):
            S.append((x, y, h))

# Size of the State Space
Ns = len(S)
print('Number of state in State Space: %d.' % Ns)
```

```
Number of state in State Space: 432.
```

2.1(b). Create your action space A = {a}. What is the size of the action space N_A ?

Action includes moving forward, backward, turning and stay still. Therefore the size of the action space N_A is 7, including:

1) No movement, No turn (0,0)
2) Going Forward, No turn (1,0)
3) Going Forward, Turn Right (1,1)
4) Going Forward, Turn Left (1,-1)
5) Going Backward, No turn (-1,0)
6) Going Backward, Turn Right (-1,1)
7) Going Backward, Turn Left (-1,-1)

```
# Action a = (move, turn)
A = [];
for move in [NO_MOVE, FW, BW]:
    if move != NO_MOVE:
        for turn in [NO_TURN, LT, RT]:
            A.append((move, turn))
    else:
        A.append((NO_MOVE, NO_TURN))

# Size of the Action Space
Na = len(A)
print('Number of action in Action Space: %d.' % Na)
```

```
Number of action in Action Space: 7
```

2.1(c). Write a function that returns the probability $p_{sa}(s')$ given inputs p_e, s, a, s' .

To find the function that returns the probability $p_{sa}(s')$, we first need to identify all cases for actions that can be done in one step (from state s to state s').

_For movement: forward with the following directions:

- If the car is pointing towards 2, 3, 4: +x direction for forward moving
- If the car is pointing towards 8, 9, 10: -x direction for forward moving
- If the car is pointing towards 11,0,1: +y direction for forward moving
- If the car is pointing towards 5, 6, 7: -y direction for forward moving

_For rotation: left, right or no turn:

- For example, if the car is pointing towards 0, turning left (counterclockwise) will change the heading to 11 and turning right will change the heading to 1. Choosing not to turn won't change the heading and it remains 0.

The robot will first rotate by +1 or -1 with probability p_e before it moves (unless it chooses not to move). Note that the car cannot pre-rotate with probability 1-2p_e, therefore the probability error p_e must be less than 0.5.

```
def angle2dir(h):
    pos = [0,0]
    if (h == 11 or h == 0 or h == 1): pos[1] = 1
    if (h == 7 or h == 6 or h == 5): pos[1] = -1
    if (h == 2 or h == 3 or h == 4): pos[0] = 1
    if (h == 10 or h == 9 or h == 8): pos[0] = -1
    return pos
```

Let i goes from 0 to 11 for 12 cases. For rotation, three cases with their probability are defined using mod function:

```
P_direction[i]= [(i, 1-2*pe, angle2dir(i)), ((i+1)%12, pe, angle2dir((i+1)%12)), ((i+11)%12, pe, angle2dir((i+11)%12)
```

For example if i =0,

P_direction[0]= [(0,1-2*pe,angle2dir(0)),(1,pe,angle2dir(1)),(11,pe,angle2dir(11)]

Which is corresponding to `# P_direction[h] = [not pre_rotate,rotate right,rotate left]`

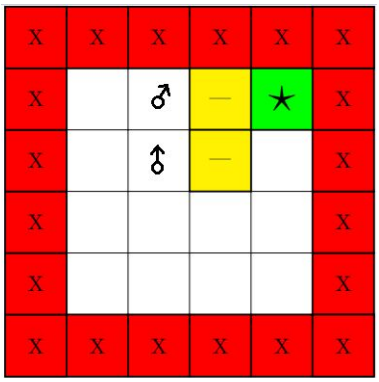
After the robot has pre-rotated, it will choose to move 1 unit in x, y, -x, or -y direction.

The last step is to calculate the probability for the next state (s') and check that if it is going to touch the border. If it tends to touch the border then it will just stay still.

```
if a[0] == FW:
    for direction in P_direction[s[2]]:
        # check if move out of grid then stay still
        xd = s[0] + direction[2][0] if ((s[0] + direction[2][0]<=L-1) and (s[0] + direction[2][0]>=0)) else s[0]
        yd = s[1] + direction[2][1] if ((s[1] + direction[2][1]<=W-1) and (s[1] + direction[2][1]>=0)) else s[1]
        hd = (direction[0] + a[1]) % 12
        P[(xd, yd, hd)] = direction[1]
```

For example, for s= (x, y, h), if h = 2 then possible moving direction can be 1, 2 or 3. Then we check all the directions to see if any of them will lead the robot to touch the border, otherwise it is possible to move or turn. The same logic is applied for going backward case. If the robot is neither turning nor moving, the probability for next state is 1.

To test for this function, we define the following inputs (here is a sample picture representing current state and next state, drawn using Paint)



```
s = (2,3,0)
s_prime = (2,4,1)
pe = 0.2
for a in A:
    p = psa_func(s, a, s_prime,pe)
    print("Action:",a)
    print("P: ",p)
```

The result is given by:

```
Action: (0, 0)
P:  0.0
Action: (1, 0)
P:  0.2
Action: (1, -1)
P:  0.0
Action: (1, 1)
P:  0.6
Action: (-1, 0)
P:  0.0
Action: (-1, -1)
P:  0.0
Action: (-1, 1)
P:  0.0
```

Recall:
No movement, No turn (0,0)
Going Forward, No turn (1,0)
Going Forward, Turn Right (1,1)
Going Forward, Turn Left (1,-1)
Going Backward, No turn (-1,0)
Going Backward, Turn Right (-1,1)
Going Backward, Turn Left (-1,-1)

This result makes sense since the highest probability (0.6) to move from state s= (2,3,0) to s'= (2,4,1) is moving forward and turning right. The second possibility is going forward without turning (since probability error is 0.2 in this case). The order actions have probability of zero since they cannot be done to reach from state s to state s'.

2.1(d). Write a function that uses the above to return a next state s' given error probability pe, initial state s, and action a. Make sure the returned value s' follows the probability distribution specified by p_{sa}

Method 1: Given input s, a, p_e, the next state s' can be found by scanning all possible cases (there are 3 cases for x, 3 cases for y, 5 cases for the heading angle, therefore the total cases are 3*3*5=45)

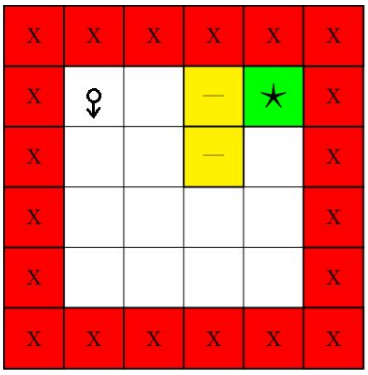
```
def next_state(s, a, pe = 0.0):
    ns = {}
    # Create Possible Next State Space
    PNS = []
    for x in [s[0]-1,s[0],s[0]+1]:
        for y in [s[1]-1,s[1],s[1]+1]:
            for h in [s[2]-2,s[2]-1,s[2],s[2]+1,s[2]+2]:
                PNS.append((x, y, h))
    for s_prime in PNS:
        p=psa_func(s,a,s_prime,pe)
        if p!=0 :
            ns[(p,s_prime)]=s_prime
    return ns
```

We test the function by choosing a current state s = (1,4,6) and p_e =0.3.

```
s = (1,4,6)
for a in A:
    ns = next_state(s, a,0.3)
    print("action:",a)
    print(list(ns.keys()))
```

The result is given by:

(here is a sample picture representing current state s = (1,4,6), drawn using Paint)



```
action: (0, 0)
[(1, (1, 4, 6))]
action: (1, 0)
[(0.3, (1, 3, 5)), (0.4, (1, 3, 6)), (0.3, (1, 3, 7))]
action: (1, -1)
[(0.3, (1, 3, 4)), (0.4, (1, 3, 5)), (0.3, (1, 3, 6))]
action: (1, 1)
[(0.3, (1, 3, 6)), (0.4, (1, 3, 7)), (0.3, (1, 3, 8))]
action: (-1, 0)
[(0.3, (1, 5, 5)), (0.4, (1, 5, 6)), (0.3, (1, 5, 7))]
action: (-1, -1)
[(0.3, (1, 5, 4)), (0.4, (1, 5, 5)), (0.3, (1, 5, 6))]
action: (-1, 1)
[(0.3, (1, 5, 6)), (0.4, (1, 5, 7)), (0.3, (1, 5, 8))]
```

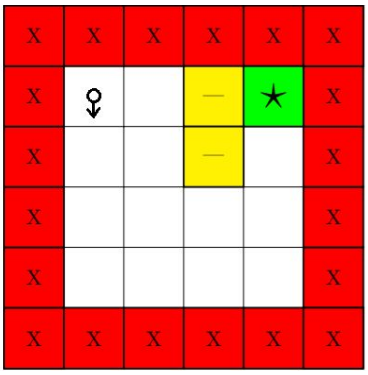
Method 2: This method uses the exact functions in part c to find all possibility for next state s', which is much faster since it does not have to scan through 30 cases like the first method.

```
def fnext_state(s, a, pe = 0.0):
```

```
def fnext_state(s, a, pe = 0.0):
    # Probability of not pre_rotate is 1-2pe, so pe <= 0.5
    if (pe > 0.5 or pe < 0):
        raise ValueError('Probability (pe) must be between 0 and 0.5')

    # Calculate the direction moving after pre_rotating for 12 case of heading
    # P_direction[h] = [not pre_rotate,rotate right,rotate left]
    # P_direction[h] = [(h,1-2*pe,[delta_x,delta_y]),(h+1,pe,[delta_x,delta_y]),(h-1,pe,[delta_x,delta_y])]

    # function return the changing in direction if go FW with the heading angle h
    def angle2dir(h):
        pos = [0,0]
        if (h == 11 or h == 0 or h == 1): pos[1] = 1
        if (h == 7 or h == 6 or h == 5): pos[1] = -1
        if (h == 2 or h == 3 or h == 4): pos[0] = 1
        if (h == 10 or h == 9 or h == 8): pos[0] = -1
        return pos

    P_direction = {}
    for i in range(12):
        P_direction[i]= [(i, 1-2*pe, angle2dir(i)), ((i+1)%12, pe, angle2dir((i+1)%12)), ((i+11)%12, pe, angle2dir((i+11)%12)

    # Calculate all possible furture state and store in P
    P = {}                # P[psa] = s'
    if a[0] == FW:
        for direction in P_direction[s[2]]:
            # check if move out of grid then stay still
            xd = s[0] + direction[2][0] if ((s[0] + direction[2][0]<=L-1) and (s[0] + direction[2][0]>=0)) else s[0]
            yd = s[1] + direction[2][1] if ((s[1] + direction[2][1]<=W-1) and (s[1] + direction[2][1]>=0)) else s[1]
            hd = (direction[0] + a[1]) % 12
            if direction[1]!=0:
                P[(direction[1],(xd, yd, hd))] = (xd, yd, hd)
```

Then we test with the same current state s = (1,4,6) and the given result is exactly the same as method 1:

```
s = (1,4,6)
for a in A:
    ns = fnext_state(s, a, 0.3)
    print("action:",a)
    print(list(ns.keys()))
```

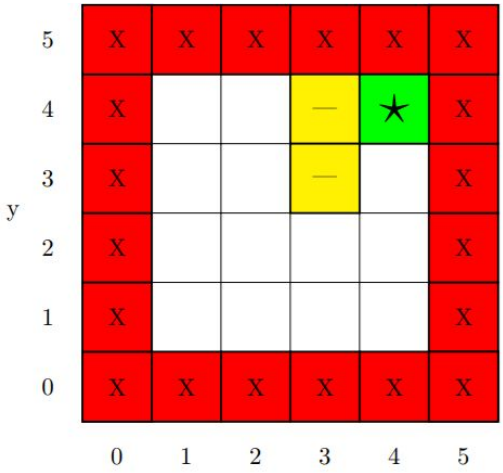


```
action: (0, 0)
[(1.0, (1, 4, 6))]
action: (1, 0)
[(0.4, (1, 3, 6)), (0.3, (1, 3, 7)), (0.3, (1, 3, 5))]
action: (1, -1)
[(0.4, (1, 3, 5)), (0.3, (1, 3, 6)), (0.3, (1, 3, 4))]
action: (1, 1)
[(0.4, (1, 3, 7)), (0.3, (1, 3, 8)), (0.3, (1, 3, 6))]
action: (-1, 0)
[(0.4, (1, 5, 6)), (0.3, (1, 5, 7)), (0.3, (1, 5, 5))]
action: (-1, -1)
[(0.4, (1, 5, 5)), (0.3, (1, 5, 6)), (0.3, (1, 5, 4))]
action: (-1, 1)
[(0.4, (1, 5, 7)), (0.3, (1, 5, 8)), (0.3, (1, 5, 6))]
```

Part III. Planning problem:

2.2(a). Write a function that returns the reward R(s) given input s.

Description: The rewards for each state are independent of heading angle (or action is taken). The border states {x = 0, x = L, y = 0, y = W} (red, marked X) have reward -100. The lane markers (yellow, marked —) have reward -10. The goal square (green, marked ⋆) has reward +1. Every other state has reward 0.



Since the reward is independent of the actions the robot takes, the reward is solely dependent on the states. Therefore the corresponding reward map is written as:

```
def reward(s):
    # Build reward map
    R_map = [(-100,-100,-100,-100,-100,-100),
             (-100,0   ,0   ,-10 , 1  ,-100),
             (-100,0   ,0   ,-10 , 0  ,-100),
             (-100,0   ,0   ,0   , 0  ,-100),
             (-100,0   ,0   ,0   , 0  ,-100),
             (-100,-100,-100,-100,-100,-100)]
    return R_map[W-s[1]-1][s[0]]
```

Part IV. Policy iteration:

2.3(a). Create and populate a matrix/array that stores the action $a = \pi_0(s)$ prescribed by the initial policy π_0 when indexed by state s.

Description: Assume an initial policy π_0 of taking the action that gets you closest to the goal square. That is, if the goal is in front of you, move forward; if it is behind you, move backward; then turn the amount that aligns your next direction of travel closer towards the goal (if necessary). If the goal is to your left or right, move forward then turn appropriately.

We first define a vector that corresponds to x,y position from the current state to goal (called direction vector). If this vector = [0,0] meaning that it has reached the goal and the action will be No Move and No Turn.

```
dir_vector = [goal[0] - s[0], goal[1] - s[1]]

# dir_vector = [0,0] means reach goal
if dir_vector == [0, 0]:
    policy[s] = (NO_MOVE, NO_TURN)
```

From the description of policy, that is, if the goal is in front of you, move forward; if it is behind you, move backward, we list all the directions for the heading that depends on the direction vector:

```
# +y direction
if s[2] in [11, 0, 1]:
    move = FW if (dir_vector[1]>=0 or dir_vector[0]==0) else BW
# -y direction
if s[2] in [5, 6, 7]:
    move = FW if (dir_vector[1]<=0 or dir_vector[0]==0) else BW
# +x direction
if s[2] in [2, 3, 4]:
    move = FW if (dir_vector[0]>=0 or dir_vector[1]==0) else BW
# -x direction
if s[2] in [8, 9, 10]:
    move = FW if (dir_vector[0]<=0 or dir_vector[1]==0) else BW
```

Similar to the direction vector for x and y, we compute an angle vector to find the angle needed to turn in order to reach closer to the goal. The vector is calculated by the difference between the current state heading and the direction vector. Since each change in one heading is 30 degree (360/12=30), the angle vector can be calculated by:

```
    th = np.arctan2(dir_vector[1], dir_vector[0])*180/np.pi
    # different between current state heading to the direction vector
    thd = s[2]*30-(90 - th)
    if (thd > 0) and (thd < 180):
        turn = LT
    elif (thd == 0) or (thd == 180):
        turn = NO_TURN
    else:
        turn = RT

    policy[s] = (move, turn)
return policy
```

2.3(b). Write a function to generate and plot a trajectory of a robot given policy matrix/array π, initial state s0, and error probability pe.

Description: Given the initial policy (i_policy), initial states (i_s), the goal (goal), and the error probability of the robot (pe), we can find the optimal actions to take by policy iteration. The optimal policy finds the best action the robot can do at this moment, and assumes continue doing the best actions.

The method trajectory in this part has two functions: generating the trajectory of the robot, and displaying the trajectory. Since the code to plot the trajectory is more of a hard-coding job, and more on knowing how to code in a particular language, most of the explanation will not be focusing on the plotting part, but on generating the trajectory. The method returns a list of states and action the robot takes to go from a given initial starting point to a given goal.

```
# 2.3b Function that generate and plot a trajectory of robot
# given initial policy, initial state, and pe
def trajectory(i_policy, i_s, goal, pe=0.0):
    # Probability of not pre_rotate is 1-2pe, so pe <= 0.5
    if (pe > 0.5 or pe < 0):
        raise ValueError('Probability (pe) must be between 0 and 0.5')
```

First, we ran through a brief scanning on the error probability to see if the input probability makes mathematical sense. Since the robot will not pre-rotation with probability equals to $1\text{-}2p_e$, and that we know that the pre-rotation probability will not be less than 0, we can find the range of the error probability to be between 0 to 0.5 inclusive ($p_e = [0, 0.5]$). However, if the error probability of the robot goes up to 0.5, it will be meaningless to compute all these analyses, since the error is too huge, it will be better to recompute the error probability or change some components.

```
    # Generate the trajectory
    traj = []
    curr_s = i_s
    while True:
        traj.append([curr_s, i_policy[curr_s]])
        if (i_policy[curr_s] == (0, 0)):
            break
        if (curr_s[0] == goal[0]) and (curr_s[1] == goal[1]):
            break
        P_ns = fnext_state(curr_s, i_policy[curr_s], pe)
        # find the best next state to go
        best_s = max(P_ns.keys())
        curr_s = P_ns[best_s]

    # Draw grid map
    #...
```

The method fnext_state () returns the probabilities of landing onto a particular next state with a given action as stated in part 2.1(d). In this function, the next state is considered to be the state with the highest probability the robot may land on after taking a specific action (found by the current policy at the current state). It is repeated until the robot arrives at the goal or current policy decided that the robot should stay still.

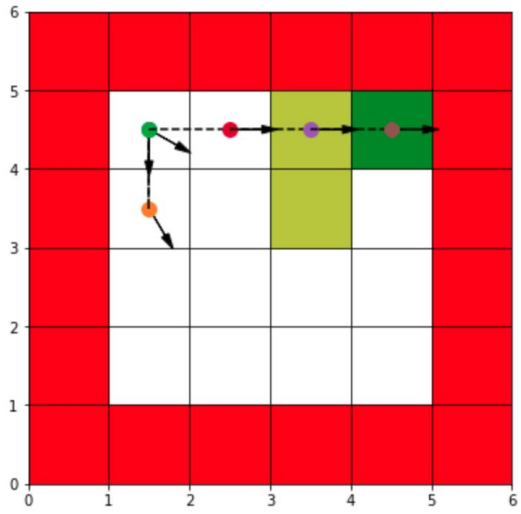
2.3(c). Generate and plot a trajectory of a robot using policy π0 starting in state x = 1, y = 4, h = 6 (i.e. top left corner, pointing down). Assume pe = 0.

```
# 2.3c Plot trajectory starting at x=1,y=4,h=6, pe=0
s0 = (1, 4, 6)
policy = init_policy()
result = trajectory(policy, s0, goal, pe=0.0)
```

Setting the initial state s0 = (1, 4, 6), and initializing the policy, we can use the trajectory function above to find the optimal path to the goal at (x = 4, y = 4). The results are shown below.

The path showed that the robot first goes down one cell and turned counterclockwise, moved backward and turned counterclockwise back to its starting point, then go all the way to the goal across the barriers (marked - and in yellow), facing to the right of the grid world with a reward of -10. The trajectory generated is not optimal since has redundant actions (going back and forth) and has a negative reward (crosses the barrier shown in light green). This is because the policy evaluation is simply based on the probability the robot may end up in a state with a given action and so, the policy chosen is not the optimal policy. The policy evaluation does not consider factors like rewards and future value. It is basically ignoring the $r(s, \pi(s), s')$ and the $\gamma V_i^{\pi}(s')$ term in the policy evaluation:

$$\gamma V_{i+1}^{\pi}(s') \leftarrow \sum_{s'} p(s, \pi(s), s')(r(s, \pi(s), s') + \gamma V_i^{\pi}(s))$$




```
Trajectory (state, action) from
(1, 4, 6) to [4, 4] is:  [[(1,
4, 6), (1, -1)], [(1, 3, 5), (-
1, -1)], [(1, 4, 4), (1, -1)],
[(2, 4, 3), (1, 0)], [(3, 4,
3), (1, 0)], [(4, 4, 3), (1,
0)]]
```

2.3(d). Write a function to compute the policy evaluation of a policy π. That is, this function should return a matrix/array of values v = V Π(s) when indexed by state s. The inputs will be a matrix/array storing π as above, along with discount factor λ.

```python
# 2.3d Policy evaluation
# evaluation will stop when the value function less than theta
# return {S: V} representing the value function.
def policy_eval(policy, reward_f, pe, discount_factor, accuracy=0.01):

    # Initial with all 0 value in V
    V = {}
    for s in S:
        V[s] = 0.0

    while True:
        delta = 0
        # evaluation for each state
        for s in S:
            v = 0
            # List all posible next state and probabilites in P_ns
            P_ns = fnext_state(s, policy[s], pe)
            for key in P_ns.keys():
                # Calculate the expected value
                ns = P_ns[key]
                v = v + list(key)[0] * (reward_f(ns) + discount_factor * V[ns])
            # Changing value of V[s]
            delta = max(delta, np.abs(v - V[s]))
            V[s] = v
        # Stop evaluating if changing less than theta
        if delta < accuracy:
            break
    return V
```

The function policy_eval() returns the value function, given the policy (policy), a reward matrix of the grid world (reward_f), probability error of the robot (pe), the discount factor γ (discount_factor), and accuracy.

This piece of code implements the policy evaluation:

$$\gamma V_{i+1}^{\pi}(s') \leftarrow \sum_{s'} p(s, \pi(s), s')(r(s, \pi(s), s') + \gamma V_i^{\pi}(s))$$

The discount factor γ is less than 1 so that it makes sure the policy will converge if the horizon of the sum is infinite.

2.3(e). What is the value of the trajectory in 2.3(c)? Use λ = 0.9.

```python
# 2.3e Trajectory starting at x=1,y=4,h=6, pe=0 with discount_factor = 0.9
pe = 0
discount_factor=0.9
accuracy = 0.001
V = policy_eval(policy,reward, pe, discount_factor, accuracy)
s0=(1,4,6)
print("The value V[(%s)] is:" % str(s0), V[s0])
```

```
The value V[((1, 4, 6))] is: -597.1163982764241
```

2.3(f). Write a function that returns a matrix/array π giving the optimal policy given a one-step lookahead on value V.

```
# 2.3f Function to calculate an optimal policy, given V, reward function, pe, discount_factor
# return optimal policy by one step lookahead
# complexity Ns*Na
def one_step_optimal(V, reward_f, pe, discount_factor):
    policy = {}
    # scan all states in state space
    for s in S:
        action_values = np.zeros(Na)
        # Look at one step ahead for possible next actions
        for i, a in enumerate(A): # i: index number, a: (move, turn)
            # List all possible next states
            P_ns = fnext_state(s, a, pe)
            for key in P_ns.keys():
                ns = P_ns[key]
                # Expected value
                action_values[i] += list(key)[0] * (reward_f(ns) + discount_factor * V[ns])

        # Find the best action
        best_action = np.argmax(action_values)
        policy[s] = A[best_action]
    return policy
```

This function one_step_optimal() computes the optimal policy with a given value (V), a reward matrix (reward_f), probability error of the robot (pe), and the discount factor (discount_factor).

One step lookahead considers all actions in the action space (Na) the robot can take and the states the robot ends in after taking the action and pick the best action such that the robot will end up with the highest reward. The policy that generates the best action is the optimal policy. It computes the best action the robot can take at the moment and assumes the robot continue to take the best action.

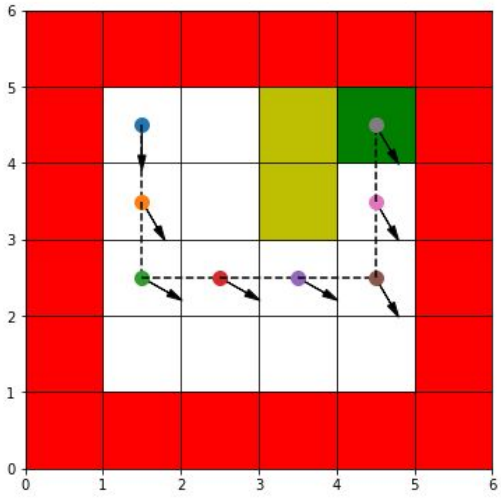
2.3(g). Combine your functions above in a new function that computes policy iteration on the system, returning optimal policy π* with optimal value V*.

```
# 2.3g Policy Iteration Algorithm
# given initial policy, reward function, pe, discount_factor
# return policy and optimal value
# note that this function have complexity of (# of iteration)*Na*Ns^2 (the times call next_state()
# the speed of calculation is depent a lot in how fast you calculate for next state.
def p_iter(policy, reward_f, pe, discount_factor):
    while True:
        # Evaluate the current policy
        V = policy_eval(policy, reward_f, pe, discount_factor)
        # Update policy using one_step_optimal function
        policy_new = one_step_optimal(V, reward_f, pe, discount_factor)
        # if new policy != old policy, continue iterate policy
        # if they equal means we found optimal policy, get out of iteration
        if policy_new != policy:
            policy = policy_new
        else:
            return policy, V
```

With the policy_eval() and one_step_optimal() functions implemented in 2.3 (d), (f), we can find the optimal policy using the one_step_optimal() function with the optimal value V* found by policy_eval() function. These two functions are called repeatedly to refine the value and policy. This process is called the policy iteration.

2.3(h). Run this function to recompute and plot the trajectory and value of the robot described in 2.3(c) under the optimal policy π^*

```
# 2.3h and 2.3i
import time

# Compute the optimal policy
start_time = time.time()
policy = init_policy()
policy_optimal, V_optimal = p_iter(policy, reward, 0.0, 0.9)

# plot the trajectory with optimal policy
s0 = (1, 4, 6)
traj = trajectory(policy_optimal, s0, goal, 0.0)
end_time = time.time()
print("Trajectory from %s to the %s is: " %(str(s0), str(goal)), traj)
print("Iteration time = %s s" %str(end_time - start_time))
```




```
Trajectory from (1, 4, 6) to the [4,
4] is:  [[(1, 4, 6), (1, -1)], [(1, 3,
5), (1, -1)], [(1, 2, 4), (1, 0)],
[(2, 2, 4), (1, 0)], [(3, 2, 4), (1,
1)], [(4, 2, 5), (-1, 0)], [(4, 3, 5),
(-1, 0)], [(4, 4, 5), (0, 0)]]
Iteration time = 5.582518100738525 s
```

As shown in the figure above, the robot takes a different path from the trajectory generated from 2.3(c). This is because the reward and the value factor is considered in the policy evaluation here. The goal for choosing the policy is to maximize the rewards of a series of actions to go from starting point to the goal. Since going through a barrier cell costs 10 points, the robot has to go around the barriers to maximize the rewards. The robot goes around the barrier and arrives at the goal with reward = 1.

2.3(i). How much compute time did it take to generate your results from 2.3(h)? You may want to use your programming language's built-in runtime analysis tool.

The computation time to generate results from 2.3(h) is around 5.58s. The computation time may vary in every computer because of the slight difference in computation power of each computer. However, the variation is not much, the computation time of our policy iteration is around 5.6 seconds.

Part IV. Value iteration:

2.4(a). Using an initial condition V (s) = 0 ∀s ∈ S, write a function (and any necessary subfunctions) to compute value iteration, again returning optimal policy $\pi*$ with optimal value $V*$.

Description: In order to find optimal policy $\pi*$ and its optimal value $V*$, we write a function value_inter() with a given value (V), a reward matrix (reward_f), probability error of the robot (pe), and the discount factor (discount_factor) and accuracy = 0.01. The function will scans through all states to find the best action value.

The function value_inter() iterates while the value of delta (our stopping condition) smaller than our given accuracy value. That function scan through all states, with each state, take all action and evaluate the action and return optimal policy with value for each action. The reward value of each action (action_values) is calculate using the reward function, discount factor (constant) and its value.

```
# List all possible next states
P_ns = fnext_state(s, a, pe)
for key in P_ns.keys():
    ns = P_ns[key]
    # Calculate the action value
    action_values[i] += list(key)[0] * (reward_f(ns) + discount_factor * V[ns])
```

Best one-step lookahead evaluate the best action that the robot have highest reward, and update the policy for that state.

```
# Best one-step lookahead
best_action_value = np.max(action_values)
best_action = np.argmax(action_values)
policy[s] = A[best_action]
```

We updated delta and set V to that state value, the loop continues iterate through all states in State Space.

```
# Calculate delta
delta = max(delta, np.abs(best_action_value - V[s]))
# Update the value function
V[s] = best_action_value
```

When the value of delta is smaller than our given accuracy, the iteration stops and return optimal policy $\pi*$ and its optimal value $V*$

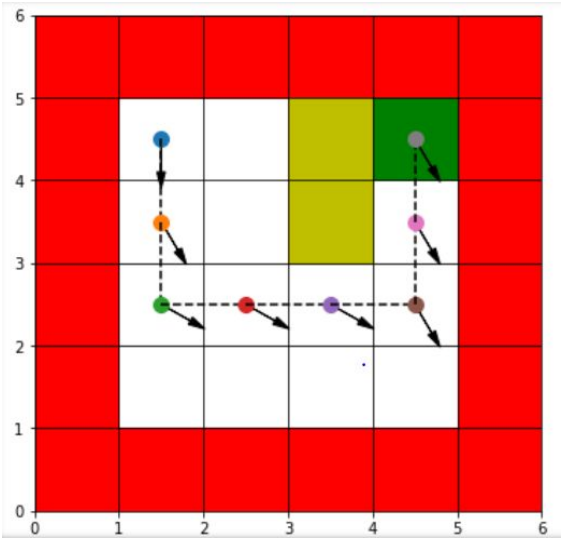
The result of this function value_inter is the same as the result of the policy_iter function in 2.3(g) but it calculates faster (about 2 times).

2.4(b). Run this function to recompute and plot the trajectory and value of the robot described in 2.3(c) under the optimal policy $\pi*$. Compare these results with those you got from policy iteration in 2.3(h).

We recompute and plot the trajectory with a current state s = (1,4,6) and p_e =0 under value iteration. The goal is at x = 4, y = 4.

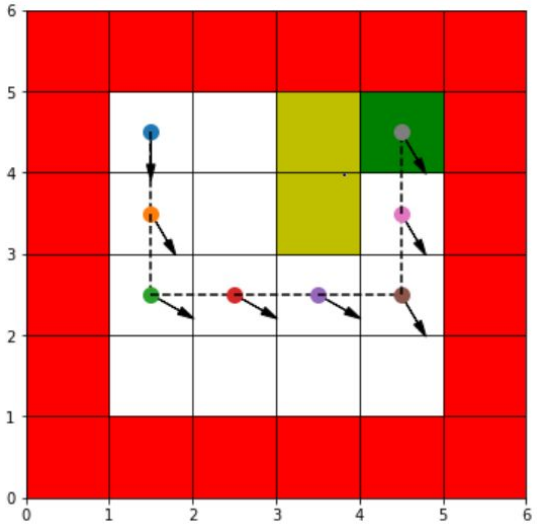
```
# Compute the optimal policy
start_time = time.time()
policy = init_policy()
policy_optimal, V_optimal = value_iter(policy, reward, 0.0, 0.9)

# plot the trajectory with optimal policy
s0 = (1, 4, 6)
traj = trajectory(policy_optimal, s0, goal, 0.0)
end_time = time.time()
print("Trajectory from %s to the %s is: " %(str(s0), str(goal)), traj)
print("Iteration time = %s s" %str(end_time - start_time))
```

We got the result plot on the left and compare with result plot from 2.3 on the right:



```
Trajectory from (1, 4, 6) to the [4, 4] is:
[[(1, 4, 6), (1, -1)], [(1, 3, 5), (1, -1)],
[(1, 2, 4), (1, 0)], [(2, 2, 4), (1, 0)],
[(3, 2, 4), (1, 1)], [(4, 2, 5), (-1, 0)],
[(4, 3, 5), (-1, 0)], [(4, 4, 5), (0, 0)]]
Iteration time = 2.9509522914886475 s
```

Compute by value iteration 2.4(b)



```
Trajectory from (1, 4, 6) to the [4, 4] is:
[[(1, 4, 6), (1, -1)], [(1, 3, 5), (1, -1)],
[(1, 2, 4), (1, 0)], [(2, 2, 4), (1, 0)], [(3,
2, 4), (1, 1)], [(4, 2, 5), (-1, 0)], [(4, 3,
5), (-1, 0)], [(4, 4, 5), (0, 0)]]
Iteration time = 5.635347604751587 s
```

Compute by policy iteration 2.3(h)

From this trajectory, we could see that each step of the car moving to reach the goal are the same for both policies, but their iteration time is different. The result paths that the robot move are the same because they both consider the best action that the car can move in order to reach the goal.

2.4(c). How much compute time did it take to generate your results from 2.4(b)? Use the same timing method as in 2.3(i).

It took t = 2.9509533914886475 seconds to generate the result while using value iteration, where for 2.3 (i) while using policy iteration took t = 5.635347604751587. Which show that compute by value iteration is about 2 times faster.

The reason for this time different is because the timing depended on how fast the policy iteration calculate for next state.

-For policy iteration: first function policy_eval() scan through all state space, Ns. Then one_step_optimal() scan through each state space and evaluate each action, Na* Ns. The policy iteration repeat have complexity of (# of iteration * Na * Ns^2)

-While for value iteration: the function value_inter() only scan through each state space and evaluate each action of that state. It will have complexity of (# of iteration * Na * Ns), which make timing of value iteration is Ns time faster (smaller) than policy iteration.

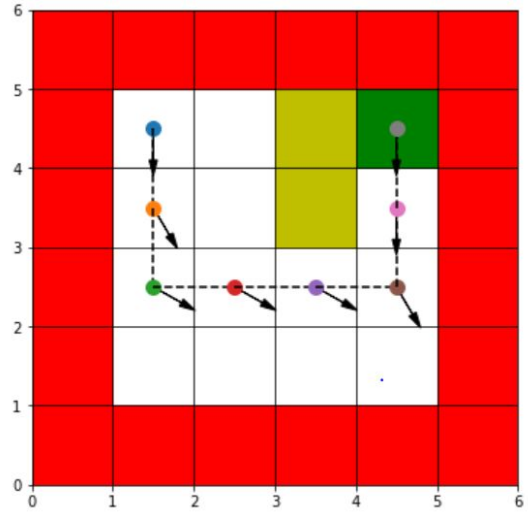
Part V. Additional scenarios:

2.5(a). Recompute the robot trajectory and value given initial conditions from 2.3(c) but with pe = 10%.

We recompute and plot the trajectory with a current state s = (1,4,6) and p_e =0.1 under using policy π*. The goal is at x = 4, y = 4.

```
# Compute optimal policy with pe=0.1
policy = init_policy()
policy_optimal, V_optimal = value_iter(policy, reward, pe = 0.1, discount_factor = 0.9)

# plot the trajectory with optimal policy
s0 = (1, 4, 6)
traj = trajectory(policy_optimal, s0, goal, 0.1)
print("Trajectory from %s to the %s is: " %(str(s0), str(goal)), traj)
```

We got the result plot:




```
Trajectory from (1, 4, 6) to the [4, 4] is:
[[(1, 4, 6), (1, -1)], [(1, 3, 5), (1, -1)],
[(1, 2, 4), (1, 0)], [(2, 2, 4), (1, 0)], [(3,
2, 4), (1, 1)], [(4, 2, 5), (-1, 1)], [(4, 3,
6), (-1, 0)], [(4, 4, 6), (0, 0)]]
```

The path for the car movement is the same, but changing in pe value leads to the change in the heading for the last 2 steps from 5 to 6 compared to question 2.4(b). This happened because error probability pe increases to 10%, which makes the rotation probability of pre-rotate off like step 5 and 6 above.

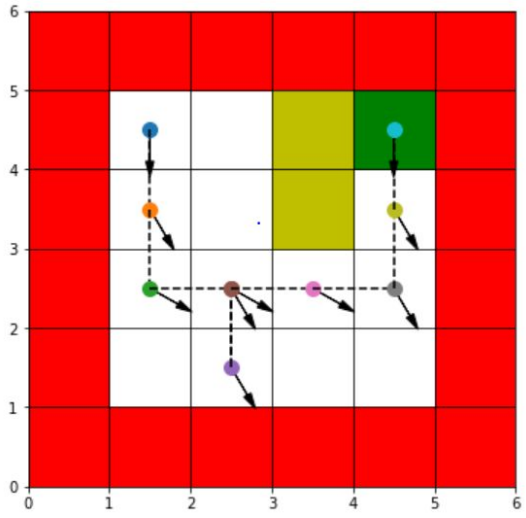
2.5(b). Assume the reward of +1 only applies when the robot is pointing straight down, e.g. h = 6 in the goal square; the reward is 0 otherwise. Recompute trajectories and values given initial conditions from 2.3(c) with pe ∈ {0, 25%}.

We recompute and plot the trajectory with a current state s = (1,4,6) and $p_e = 0.25$ under using policy π*. The goal is at x = 4, y = 4, reward = +1 when robot pointing straight down (h = 6 only)

We set the condition for reward:

```
if (s[0] == 4) and (s[1] == 4) and (s[2] == 6):
    return 1;
```

The compute the trajectory, the result plot is:




```
Trajectory from (1, 4, 6) to the [4, 4] is:  [[(1, 4, 6),
(1, -1)], [(1, 3, 5), (1, -1)], [(1, 2, 4), (1, 1)], [(2,
2, 5), (1, 0)], [(2, 1, 5), (-1, -1)], [(2, 2, 4), (1,
0)], [(3, 2, 4), (1, 1)], [(4, 2, 5), (-1, 0)], [(4, 3,
5), (-1, 1)], [(4, 4, 6), (0, 0)]]
```

2.5(c). Qualitatively describe some conclusions from these scenarios.

When the reward = +1 only at the position the car point straight down (h = 6) at goal square, the robot needs more steps to reach the goal than 2.4(b) because the error probability pe increases to 25%. Higher error probability makes the rotation probability of pre-rotation off and the robot cannot evaluate the best action (starting from step 3 in this case). It causes the robot to move off the path and need more actions to adjust/correct its steps to reach the goal.

Conclusion

Markov Decision Processes (MDP) helps us to be able to find the optimal path (with maximum reward) with a given starting point and the goal. This lab shows step by step to build up a complete MDP code and analysis, including the policy iteration and the value iteration. Both iterations give the same result in finding the optimal path.

However, even though both iterations methods provide the same results, the computation time for the methods differ due to the complexity of the methods. Value iteration is faster than policy iteration because it has a complexity of # of iteration * Na * Ns, where as policy iteration has a complexity of # of iteration * Na * Ns^{2}, even though we are able to decrease the number of iterations in finding the next state of the policy evaluation.

We were able to decrease the number of iterations in finding the next states by considering the robot's initial state, its possible actions and the possible states it may end in. In 2.1(d), we replaced next_state (), which updates the states and their probability distribution of every single state, with fnext_states (), which only compute the probability distributions of states that the robot may end up in, considering the actions the robot can take. Since in our action space, the only allowed actions are going forward, backward and turning left or right by one step (refer to 2.1 (b)), the robot can only locate in its initial states or states surrounding its initial states for the next unit of time. There is no point in analyzing the states that are 2 cells across its initial states. This way, we can reduce the number of cases from 432 to 45.

However, even with reduction of cases, MDP still takes a lot of time. Within a grid world of 6*6 and an action space of 7, the best computing time still takes up at least 2 seconds. If MDP has to be applied to the real world, we have to first discretize the continuous real world state space, time and actions. This means that the size of action space and state space may go up to hundreds or thousands. Therefore, the computation time will take hours. This will be impractical for real-time use. However, using different coding style and methods, like parallel computing (some research has been working on it) and use of GPU may greatly reduce the computing time. But that is beyond the scope of this lab.

Full Code link:
https://github.com/domlocnguyen/EE183DA/blob/master/LAB3/Lab3_tiger.ipynb

Reference
https://people.eecs.berkeley.edu/~pabbeel/cs287-fa15/slides/lecture2-mdps-exact-methods.pdf
https://people.eecs.berkeley.edu/~pabbeel/cs287-fa15/

https://en.wikipedia.org/wiki/Markov_decision_process
https://link.springer.com/chapter/10.1007/978-3-642-03095-6_45