Qiong, Zhuyun, Hailin
October 21, 2019

Lab 1 Writeup

EE209AS
Computation Robotics

1 MDP System

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

We are using coordinates (x,y) and heading direction h for state space, so for every state s=(x,y,h). For every point, there are totally 12 different directions.

Algorithm 1: Create State Space S

```
# S = {s}, s = (x, y, h) with coordinates (x, y) and heading direction h.
S = []
for x in range(L):
    for y in range (W):
        for h in range(12):
            S.append((x, y, h))
```

The size of the state space $N_S = 12 * L * W$.

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

We are using two steps for action space. The first one is move forwards or backwards, the second one is turn left or turn right or not turn.

Algorithm 2: Create Action Space A

```
# A = {a}, a = (motion, turn)
# motion_list: [STILL = 0, FORWARDS = 1, BACKWARDS = -1]
# turn_list: [NOT_TURN = 0, TURN_LEFT = -1, TURN_RIGHT = 1]
A = [];
for motion in motion_list:
    if motion != STILL:
        for turn in turn_list:
            A.append((motion, turn))
    else:
        A.append((STILL, NOT_TURN))
```

The size of the action space $N_A = 7$.

1(c). Write a function that returns the probability $p_{sa}(s')$ given inputs error probability p_e, initial state s, action a, next state s'.

Algorithm 3: Pseudocode of the Probability $p_{sa}(s')$

```
def direction(heading):
    Transform 12 headings to the nearest cardinal direction
    return [x, y]

def err_prob(heading, pe):
    Calculate probability distribution of  headings after error pre-rotating
    return err_headings

def p_sa(s, a, s_, pe):
    1. Pre-rotate probability pe validity check
```

```
    2. Check if the action a is stay still
    3. For any possible heading with probability (using function err_prob),
       check if the expected new state s_p move off the grid after action a
    4. If the given state s_ == expected new state s_p, return the probability
        of being in the expected new state s_p
    5. Otherwise, return 0
```

Transition probability $P = \{p_{sa}(s')\} = \{p(s'|s,a)\}$ satisfies:

$$\sum_{s' \in S} p_{sa}(s') = 1.$$

For a certain state s and action a, there are at most three possible s': rotate by +1 then do action a, rotate by -1 then do action a and just do action a, with the probability of p_e, p_e and $1 - 2 * p_e$.

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

Algorithm 4: Function of Possible New States s' with Probability

```
def next_state(s, a, pe):
    p={}        # The dictionary of all possible new states with probabilities
    for state in S:                          # For any state in state space S,
        if p_sa(s, a, state, pe) > 0:        # if the probability is positive,
            p[state] = p_sa(s, a, state, pe) # add to the result dictionary p
    return p
```

2 Planning problem

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

The rewards for each state are independent of heading angle. The red border states: $x = 0; x = L; y = 0; y = W$ have reward -100. The yellow lane markers have reward -10. The green goal square has reward +1. Every other state has reward 0.

Algorithm 5: Reward Function

```
def reward(s):
    if s[0] == 0 or s[0] == L - 1 or s[1] == 0 or s[1] == W - 1:
        return -100     # Border states (Red, marked X)
    elif s[0] == 3 and s[1] in [4, 5, 6]:
        return -10      # Lane Markers (Yellow, marked --)
    elif s[0] == 5 and s[1] == 6:
        return 1        # Goal state (Green, marked *)
    else:
        return 0        # Every other state has reward 0
```

3 Policy iteration

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.

The initial policy aims to approximately getting the robot closest to the goal square. If the goal is in front of the robot, move forwards; if it is behind the robot, move backwards. After moving, if the target is in the left front or the right back of the robot , turn left; if the target is in line with the robot, do not turn; else turn right.

Algorithm 6: Pseudocode of Action Storage

```
# Initialize a policy:
def policy_init(S):
    # State Space S = {s}
    # Return Policy: A dictionary of all policies
    policy = {}
    for state in S:
       # Get the vector from the state to the goal
       direction_vector = Goal location - state location

       # Already reached goal
       if direction_vector = [0, 0]
       policy[s] = (STILL, NOT_TURN)
       continue

       Compute the moving direction:
       Check the target location w.r.t. the robot position.
       Case 1. Heading +x (h in [2, 3, 4]):
       if the target is on the right, robot moves FORWARDS;
       else move BACKWARDS.
       Case 2. Heading -x (h in [8, 9, 10]):
       if the target is on the left, robot moves FORWARDS;
       else move BACKWARDS.
       Case 3. Heading +y (h in [11, 0, 1]):
       if the target is in the front, robot moves FORWARDS;
       else move BACKWARDS.
       Case 4. Heading -y (h in [5, 6, 7]):
       if the target is in the back, robot moves BACKWARDS;
       else move FORWARDS.

       Evaluate the turning direction using the angle of the direction_vector.
       Case 1: if the target is in the front left or the back right of the
          robot, turn left.
       Case 2: if the target is in line with the robot, do not turn, else turn
           right.
       return turn

       policy[s] = (motion, turn)

    return policy
```

3(b). Write a function to generate and plot a trajectory of a robot given policy matrix/array π, initial state s_0, and error probability p_e.

The function follows these steps:

1. Generate a trajectory using weighted probability given the policy *pi*.
2. Create a $8 * 8$ grid with red wall, yellow lane markers and the green target.
3. Use a circle to represent the robot and an arrow to represent the direction that the robot faces.

Algorithm 7: Pseudocode of Generating Trajectory

```
def generate_trajectory(policy, s, pe = 0, show = True):
    1. while robot is not at target:
        Get next states according policy and probabilities, choose next state
           according to their probabilities
        Use a list to restore the trajectory
    2. Plot the 8*8 Grid world
    3. Plot the green goal, yellow markers and red walls
    4. Plot the start state
    5. Plot all passing states
    6. Plot the grid world and the directory
```

3(c). Generate and plot a trajectory of a robot using policy π_0 starting in state $x = 1, y = 6, h = 6$ (i.e. top left corner, pointing down). Assume $p_e = 0$.

Algorithm 8: Generate and Plot Trajectory

```
generate_trajectory(policy_init(S), (1, 6, 6), 0)
```

The trajectory is shown in Figure 1.

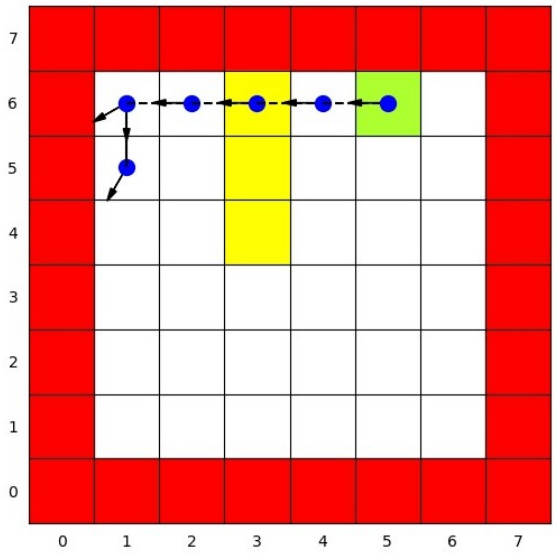

Figure 1: Trajectory of the robot using policy π_0 starting in state (1, 6, 6).

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 $v = V^{\pi}(s)$ when indexed by state s. The inputs will be a matrix/array storing π as above, along with discount factor λ.

we can calculate evaluation V of a policy π by following equation:

$$V^{\pi}(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s), s') + \lambda V^{\pi}(s')]$$

Algorithm 9: Pseudocode of Value Function

```
def policy_eval(S, V, policy, reward, pe = 0, discount_factor = 1, threshold =
    0.001):
    1. Initialize V to all 0 if not claimed
    2. for s in S: calculator V[s] using above function
    3. When V[s] does not change, the iteration is over
```

3(e). What is the value of the trajectory in 3(c)? Use $\lambda = 0.9$.

Algorithm 10: Value of Trajectory in 3(c)

```
V = policy_eval(S, policy_init(S), reward, 0, 0.9, 0.001)
s0 = (1, 6, 6)
# Trajectory value
print (V[s0])
```

The value of the trajectory in 3(c) is -1.393.

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

We could update the policy π by following equation:

$$\pi^*(s) = \arg\max_{a \in A} \Big[\sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^*(s')] \Big]$$

Algorithm 11: Pseudocode of giving the optimal policy by one-step look ahead

```
def policy_one_step_lookahead(S, V, reward, pe = 0, discount_factor = 1):
    for s in S:
        1. Calculate V[s] for every possible action a
        2. Choose the action a which maximizes V[s]
        3. Update policy
```

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^*.

Now, we do policy iteration. The process is as follows:

1. Calculate V of the policy
2. Update policy by one-step look ahead
3. Repeat 1 and 2 until the policy does not change any more

Algorithm 12: Pseudocode of policy iteration

```
def policy_iteration(S, V, policy, reward, pe = 0, discount_factor = 1):
    while policy changes:
        1. Calculate V for the policy
        2. Update policy by one step look ahead
```

3(h). Run this function to recompute and plot the trajectory and value of the robot described in 3(c) under the optimal policy π^*.

Algorithm 13: Plot the trajectory and value of the robot based on policy iteration

```
def policy_iteration_timing(reward = reward, pe = 0):
    # Keep track of the compute time.
    start = time.time()
    # Optimal_policy calculation
    optimal_policy, optimal_V = policy_iteration(S, {}, policy_init(S), reward
        , pe, 0.9)
    end = time.time()
    print("runtime is",end-start)
    # Recompute and plot the trajectory and value of the robot under optimal
        policy
    s0 = (1, 6, 6)
    trajectory = generate_trajectory(optimal_policy, s0, pe)
    print("the value of trajectory is", V[s0])
```

The trajectory is shown in Figure 2.

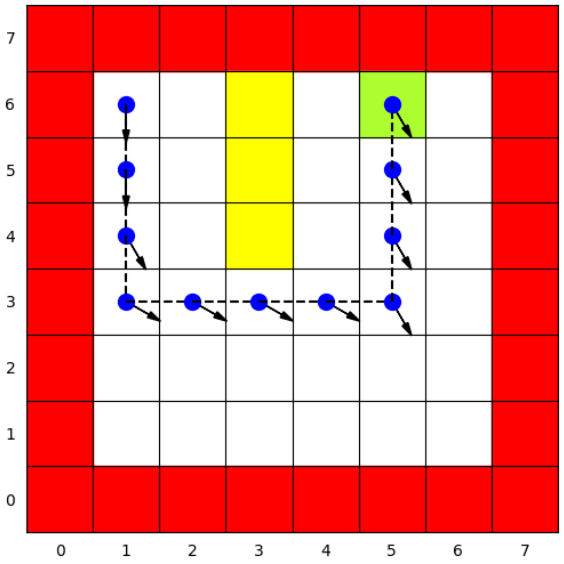

Figure 2: Trajectory of the robot using optimal policy π^* based on policy iteration starting in state (1, 6, 6).

The value of the trajectory is 3.874.

3(i). How much compute time did it take to generate your results from 3(h)? You may want to use your programming language's built-in runtime analysis tool.

The compute time is about 635 seconds, with $threshold = 0.01$.

4 Value iteration

4(a). Using an initial condition $V(s) = 0 \ \forall s \in S$, write a function (and any necessary subfunctions) to compute value iteration, again returning optimal policy π^* with optimal value V^*.

We could update the value function $V(s)$ by the following equation:

$$V^*(s) = \max_{a \in A} \Big[\sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^*(s')] \Big]$$

Now, we do value iteration. The process is as follows:

1. Choose action to maximize value function $V(s)$
2. Repeat until the value function $V(s)$ does not change any more

Algorithm 14: Pseudocode of value iteration

```
def value_iteration(S, policy, reward, pe = 0, discount_factor = 1, threshold
    = 0.01):
     Initialize Values to V(s) = 0
     while V changes:
         1. Choose action to maximize V
         2. Update policy
```

4(b). Run this function to recompute and plot the trajectory and value of the robot described in 3(c) under the optimal policy π^*. Compare these results with those you got from policy iteration in 3(h).

Algorithm 15: Value Iteration

```
def value_iteration_timing(reward = reward, pe = 0):
    # Runtime analysis.
    start = time.time()
    # Optimal_value calculation
    optimal_policy, optimal_V = value_iteration(S, policy_init(S), reward, pe,
        discount_factor = 0.9)
    end = time.time()
    print("runtime is", end-start)

    # Recompute and plot the trajectory and value of the robot under optimal
       policy
    s0 = (1, 6, 6)
    trajectory = generate_trajectory(optimal_policy, s0, pe)
    print("the value of trajectory is", V[s0])
```

The trajectory is shown in Figure 3.

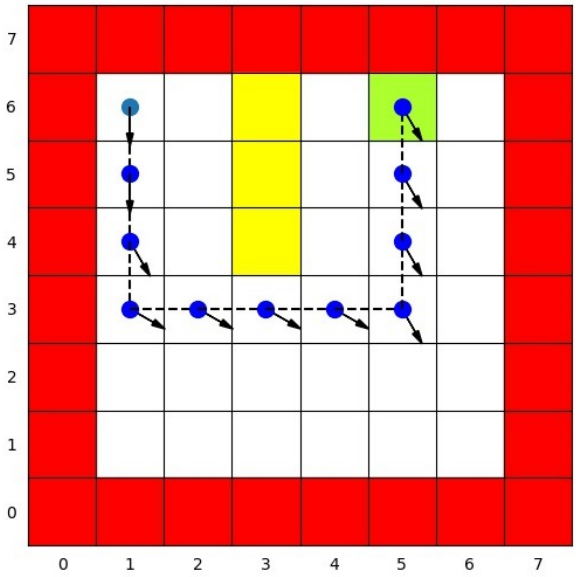

Figure 3: Trajectory of the robot using optimal policy π^* based on value iteration starting in state (1, 6, 6).

From Figure 3, We find out that the trajectory is the same as the one from policy iteration (Figure 2). The reason is that policy iteration and value iteration converge to the same policy after enough iterations.

The value of the trajectory is 3.811. Compared with the value of 3.874 from policy iteration, it seems that value iteration gives a more accurate result since it requires value function to converge as well as the policy itself.

4(c). How much compute time did it take to generate your results from 4(b)? Use the same timing method as in 3(i).

The compute time to generate results from 4(b) is about 1196 seconds, with $threshold = 0.01$, which is about twice longer than the compute time in 3(i).

5 Additional scenarios

5(a). Recompute the robot trajectory and value given initial conditions from 3(c) but with $p_e = 25\%$.

The recomputed trajectory given $p_e = 25\%$ is shown in Figure 4 (based on policy iteration), and Figure 5 (based on value iteration).

The values of trajectory are 1.829 from policy iteration and 1.788 from value iteration.

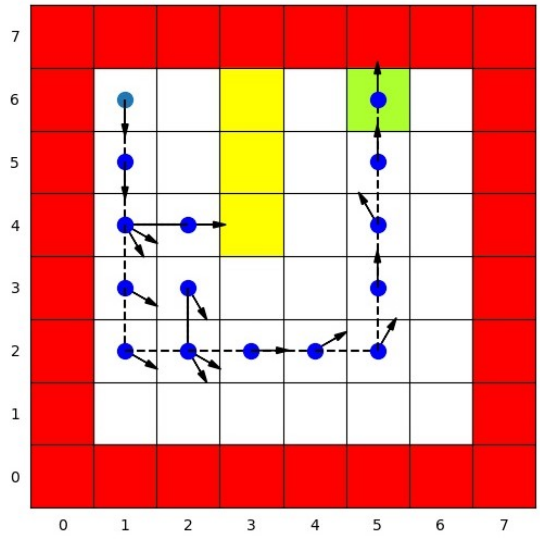

Figure 4: Trajectory of the robot based on policy iteration, using policy π^* starting in state (1, 6, 6) with $p_e = 25\%$.

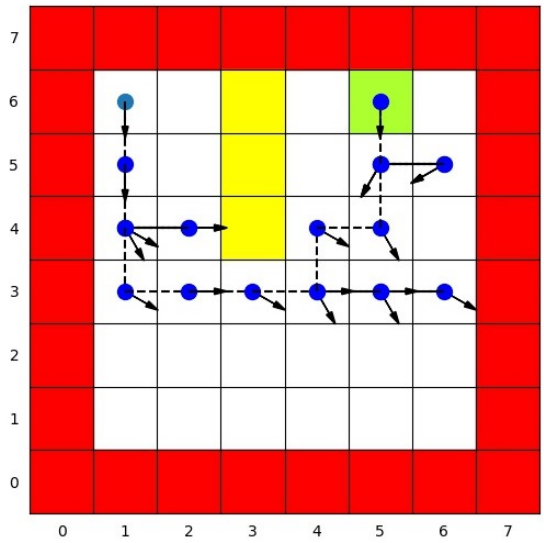

Figure 5: Trajectory of the robot based on value iteration, using policy π^* starting in state (1, 6, 6) with $p_e = 25\%$.

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 3(c) with $p_e \in \{0, 25\%\}$.

Algorithm 16: New Reward

```
def reward_new(s):
    # Current state s = (x, y, h)
    # Return: rewards of the current state

    # Border states (Red, marked X)
    if s[0] == 0 or s[0] == L - 1 or s[1] == 0 or s[1] == W - 1:
        return -100

    # Lane Markers (Yellow, marked --)
    elif s[0] == 3 and s[1] in [4, 5, 6]:
        return -10

    # Goal state (Green, marked *)
    elif s[0] == GOAL[0] and s[1] == GOAL[1] and s[2] == 6:
        return 1

    # Every other state has reward 0
    else:
        return 0
```

The recomputed trajectories with $p_e = 0$ are shown in Figure 6 (based on policy iteration) and Figure 7 (based on value iteration).

The recomputed values are 3.874 from policy iteration and 3.811 from value iteration.

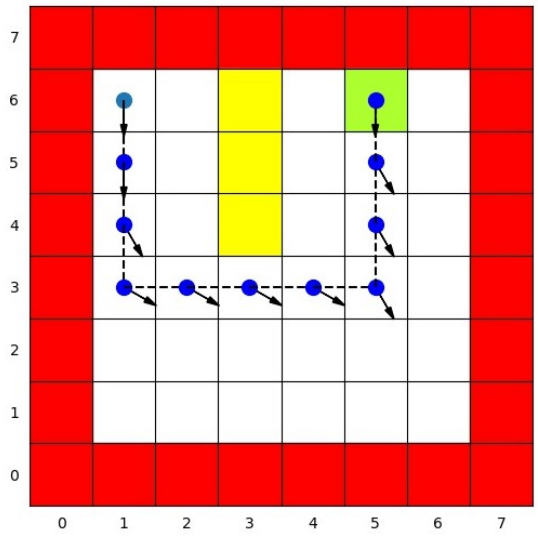

Figure 6: Trajectory with new reward function, based on policy iteration, using policy π^* starting in state (1, 6, 6) with $p_e = 0$.

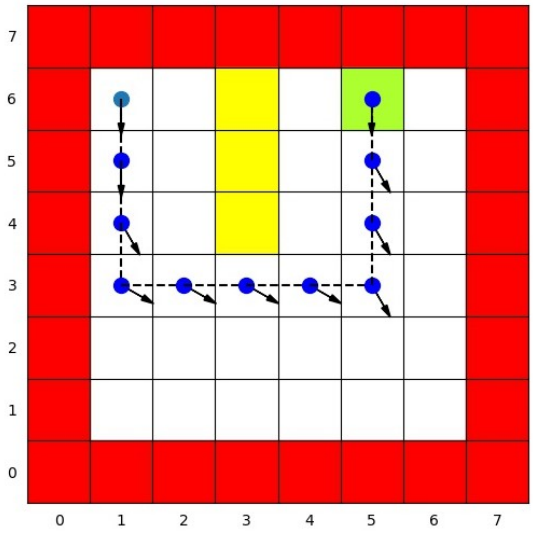

Figure 7: Trajectory with new reward function, based on value iteration, using policy π^* starting in state (1, 6, 6) with $p_e = 0$.

The recomputed trajectories with $p_e = 25\%$ are shown in Figure 8 (based on policy iteration) and Figure 9 (based on value iteration).

The recomputed values are 1.328 from policy iteration and 1.297 from value iteration.

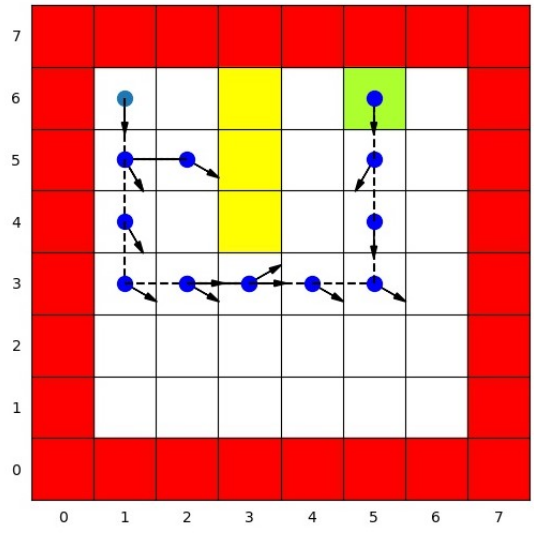

Figure 8: Trajectory with new reward function, using policy iteration, using policy π^* starting in state (1, 6, 6) with $p_e = 0.25\%$.

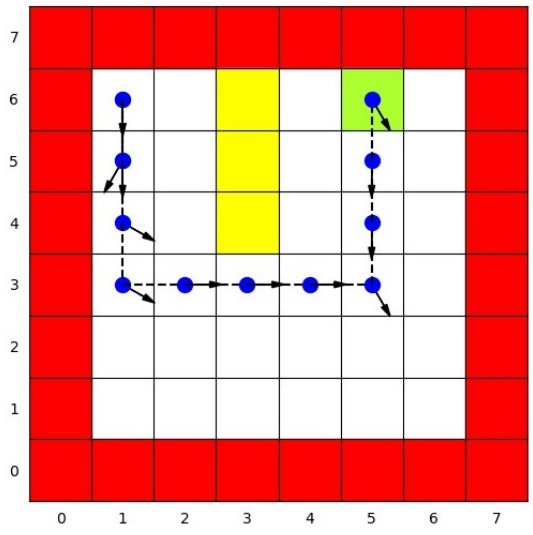

Figure 9: Trajectory with new reward function, using value iteration, using policy π^* starting in state (1, 6, 6) with $p_e = 0.25\%$.

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

All the results are shown in the Table 1.

Table 1: Value and Compute Time Results for 8 experiments

Iteration type	p_e	Reward Function	$V[s0]$	Compute Time/s
policy	0	+1 at goal square	3.87	635
value	0	+1 at goal square	3.81	1196
policy	0.25	+1 at goal square	1.83	1447
value	0.25	+1 at goal square	1.79	1267
policy	0	+1 at goal square and $h = 6$	3.87	665
value	0	+1 at goal square and $h = 6$	3.81	1255
policy	0.25	+1 at goal square and $h = 6$	1.33	1379
value	0.25	+1 at goal square and $h = 6$	1.30	1332

The qualitative conclusions are as follows:

1. $V[s]$ for value iteration is always smaller than that of policy iteration. The reason may be that compared to the threshold in policy iteration, the same threshold 0.01 in value iteration means a more rigorous convergence requirement.

2. When $p_e = 0$, the compute time of policy iteration is shorter than that of value iteration. The reason is that, without error, there will only be one possible next state s' for any certain state s and action a given current policy, so the search, update and iteration of policy is quicker than those of value.

3. When $p_e = 0.25$, the compute time of value iteration is shorter than that of policy iteration. The reason is that, there will be two or three possible next states s' for any certain state s and action a given current policy, so it takes more time to calculate and choose an optimal policy in each round of policy iteration, thus the complexity of policy iteration raises quickly.

4. When $p_e = 0$, the new reward function has the same $V[s0]$ value as the old reward function under corresponding experiment settings. The reason is that the optimal policy generated from old policy iteration happens to be the one that the robot will direct in $h = 6$ in the goal square. It may not remain the same for other starting states.

5. When $p_e = 0.25$, the new reward function has smaller $V[s0]$ value than that of old reward function. The reason is that the $reward = 1$ space is smaller according to the new reward function, so the probability of getting reward is smaller for the robot.

Supplementary Resources

1. Initial code: https://github.com/Qiong-Hu/Computational_Robotics/tree/master/lab_1/MDP.py
2. Resulting images: https://github.com/Qiong-Hu/Computational_Robotics/tree/master/lab_1/result

Reference

1. Lecture content of ECE209AS
2. Github link: https://github.com/Jingwen-Zhang-Aaron/Computational-Robotics/blob/master/pset2-Policy-Value-Iteration/HW2.ipynb
3. M. Littman et al. "On the Complexity of Solving Markov Decision Problems", https://arxiv.org/pdf/1302.4971.pdf