CS 533 Homework 4
Reinforcement Learning for Optimal Parking

Eugene Seo

05-05-2017

Part I: Designing the MDPs

I designed three parts to simulate Parking MDP: MDP, Agent, Simulator

- MDP Class:
  It generates a MDP model given parameters such as n, rewards, probabilities. It also provides the initial state or the next state to an agent based on its MDP model.
- Agent Class:
  It represents an agent playing around in the parking simulation. It also can learn the optimal parking behavior through many trials.
- Simulator:
  It creates MDPs and agents using MDP and Agent classes, and let agents learn in each MDP model through many attempts.

  Run simulation: python Simulator.py

Algorithm 1 MDP Class

```
class MDP:
    def __init__(self, n):
    def make_MDP(self, filename, rewards, probabilities):
    def build_MDP(self, n, penalty_driving, penalty_handicap, penalty_collision,
                        best_reward, prob_avail_handicap, prob_T, filename):
    def write_MDP(self, filename, T_drive, T_park, T_exit, R):
    def init_state(self):
    def next_state(self, s, a):
```

Algorithm 2 Agent Class

```
class Agent:
    def __init__(self, mdp):
    def init_state(self, init_s):
    def set_policy(self, policy):
    def GLIE_policy(self, s, t):
```

Algorithm 3 Simulator

```
def main():
    myMDPs = buildTwoMDPs()
    """ PART 2 """
    max_trial = 1000
    basic_policies_simulation(myMDPs, max_trial)
    """ PART 3 """
    max_trial = 50
    learning_num = 500
    reinforcement_learning(myMDPs, max_trial, learning_num)
```

For my parking simulations, I built two MDPs as follows:

Algorithm 4 myMDPs

```
def buildTwoMDPs():
    n = 10
    penalty_handicap = -10
    penalty_collision = -100
    prob_avail_handicap = 0.9
    prob_T = 5.0

    """ 1st set of parameter values """
    penalty_driving = -1
    best_reward = 100

    """ 2nd set of parameter values """
    penalty_driving = -10
    best_reward = 10
```

- MDP1: This MDP is designed to have a less cost of driving (-1) and high rewards for the closest parking plots except the handicap spots (the highest parking reward is 100.). Thus, the optimal policy would be parting in the places in front of the store except A1 and B1.

- MDP2: This MDP is designed to have a high cost of driving (-10) and less rewards for the closest parking plots except the handicap spots (the highest parking reward is 10.). Thus, the optimal policy would be to park right away once the agent finds the available place no matter the distance from the store.

Part 2: Policy Simulation

Heres are base policies I designed for baselines of parking performance.

1. Random Policy:

I randomly select the PARK action with a probability p and DRIVE with probability 1 - p. I set p = 0.2 in my simulation. Since it could park everywhere, even making collision, it would be the worst policy.

Algorithm 5 Random Policy

```
def policy_generator_1(state_size, T, R):
    prob_park = 0.2
    policy = np.random.choice(2, state_size, p=[1-prob_park, prob_park])
    return policy
```

2. Partial Random Policy:

This policy takes the DRIVE action whenever O is TRUE, otherwise it randomly selects the PARK action with a probability p and DRIVE with probability 1 - p. I set p = 0.2 in my simulation. It prevents an agent from collision, but it's still not the best policy since it does not count any effective way to park without considering the reward of parking and the cost of driving.

Algorithm 6 Partial Random Policy

```
def policy_generator_2(state_size, T, R):
    prob_park = 0.2
    policy = np.zeros([state_size/4,4])
    policy[:, 0:2]= np.random.choice(2, state_size/2, p=[1-prob_park, prob_park]).reshap
    policy = policy.reshape(1,state_size)[0]
    return policy
```

3. My Simple Policy 1:

I designed the first simple policy as assigning the PARK action into the top 20% of front spots. Otherwise, it takes the DRIVE action especially in the farthest spots. Thus, this policy would fit with myMDP1, which have high reward in the front spots and the less cost of driving.

Algorithm 7 My Simple Policy 1

```
def policy_generator_3(state_size, T, R): # my policy
    ratio = (state_size/4 - 2)/5
    policy = np.zeros([state_size/4,4])
    #policy[2:4, 0] = 1 # policy 1
    policy[2:2+ratio, 0] = 1
    policy[policy.shape[0]-ratio:policy.shape[0], 0] = 1
    policy = policy.reshape(1,state_size)[0]
    return policy
```

4. My Simple Policy 2

In the second design of policy, I assigned the PARK action into every available spot which is not occupied except A1 and B1. Thus, this policy would fit with myMDP2, which have a high cost of driving and less reward of front parking.

Algorithm 8 My Simple Policy 2

```
def policy_generator_4(state_size, T, R): # my policy
    policy = np.zeros([state_size/4,4])
    policy[2:, 0] = 1 # policy 2
    policy = policy.reshape(1,state_size)[0]
    return policy
```

• Policy Simulation

I run the simulator with four baseline policies for both my MDPs. I run 100 trials of the policy and limited the step of each trial up to 100 steps. Below are the results from the simulation.

Policy	myMDP1	myMDP2
Random policy	-13.7980	-97.1944
Partial random policy	24.1406	-37.2679
My policy 1	32.0883	-43.1916
My policy 2	26.6502	-14.4402

Table 1: The average reward from Q-learning in each MDPs

As we can see in the table 2, my first policy, which is optimally designed for myMDP1, generate the highest reward among all baseline policies in the myMDP1 model. On the other hand, my second policy, which is designed for myMDP2, shows the best performance among policies in myMDP2. It proves that my policies are well designed for each model. The random policy shows the worst performance and the partial random policy improve the worst performance as avoiding high negative penalties caused by the collision, but it's still not the optimal policy.

Algorithm 9 Baseline Simulation

```
def main():
     """ PART 2 """
    max_trial = 1000
    max_steps = 100
    basic_policies_simulation(myMDPs, max_trial, max_steps)

def basic_policies_simulation(myMDPs, max_trial, max_steps):
    for myMDP in myMDPs:
        myAgent = agent.Agent(myMDP) policy_list.append(policy_generator_1(myMDP.state_s
        policy_list.append(policy_generator_2(myMDP.state_size, myMDP.T, myMDP.R))
        policy_list.append(policy_generator_3(myMDP.state_size, myMDP.T, myMDP.R))
        policy_list.append(policy_generator_4(myMDP.state_size, myMDP.T, myMDP.R))
        for policy in policy_list:
            myAgent.set_policy(policy)
            avg_reward = evaluation(myMDP, myAgent, max_trial, max_steps)

def evaluation(myMDP, myAgent, max_trial, max_steps):
    for i in range(0,max_trial):
        reward = simulation(myMDP, myAgent, max_steps)
        reward_list.append(reward)
    return np.average(reward_list)

def simulation(myMDP, myAgent, max_steps):
    myAgent.init_state(myMDP.init_state())
    myAgent.a = myAgent.policy[myAgent.s]

    trial_num = 1
    while(myAgent.a != 1 and trial_num < max_steps):
        myAgent.reward = myAgent.reward + myMDP.R[myAgent.s]
        myAgent.s = myMDP.next_state(myAgent.s, myAgent.a)
        myAgent.a = myAgent.policy[myAgent.s]
        trial_num = trial_num + 1

    return myAgent.reward
```

Part 3: Reinforcement Learning

I implemented a reinforcement learning agent based on the Q-learning algorithm.

- ### Explore/Exploit Policy

The agent learns the parking behavior throughout many trials. To effectively let the agent learn, I implemented an explore/exploit policy called ε-greedy exploration.

Algorithm 10 GLIE policy - ε-greedy exploration

```
def GLIE_policy(self, s, t):
    op = np.random.choice(2, 1, p=[1/t, 1-1/t])
    if op == 0: # random
        a = np.random.choice(self.action_size, 1)
    else: # greedy
        a = np.argmax(self.Q[s])
    return a
```

At the first of learning, my explore/exploit policy tries to generate random actions to explore all the possibilities. As leaning goes on, the policy tends to select the greedy policy to exploit the environment after enough exploration.

- ### Q-learning Algorithm

Here is Q-learning I implemented. I update Q function of an agent whenever the agent takes an action from the current state to the next state. Below equation shows the simple form of Q function update in my code.

$$Q[s,a] = Q[s,a] + lr * (R[s] + beta * max(Q[s']) - Q[s,a])$$

where s is the current state, s' is the next state, a is the action, and lr is the learning rate.

I set learning rate 0.9, and beta 1. I limited the number of steps up to 100, and learn the agent 500 times to measure the performance of the learned agent. Each learning time consists of 10 trials, and I measured the performance of the learned greedy policy after 1000 trials.

Algorithm 11 Q-learning Algorithm

```
def Q_learning(myMDP, myAgent, max_steps, T):
    lr = 0.9
    beta = 1

    # step 2. take action from explore/exploit policy giving new state s'
    myAgent.a = myAgent.GLIE_policy(myAgent.s, T)
    while(myAgent.a != 1 and trial_num < max_steps):
        myAgent.next_s = myMDP.next_state(myAgent.s, myAgent.a)
        # step 3. perform TD update
        myAgent.Q[s,a] = myAgent.Q[s,a] + lr *
        (myMDP.R[s] + beta * max(myAgent.Q[myAgent.next_s[0]]) - myAgent.Q[s,a])
        myAgent.s = myAgent.next_s
        myAgent.a = myAgent.GLIE_policy(myAgent.s, T)
        trial_num = trial_num + 1

    if trial_num < max_steps:
        myAgent.next_s = myMDP.next_state(myAgent.s, myAgent.a)
        myAgent.Q[s,a] = myAgent.Q[s,a] + lr *
        (myMDP.R[s] + beta * max(myAgent.Q[myAgent.next_s[0]]) - myAgent.Q[s,a])

        myAgent.s = myAgent.next_s
        myAgent.a = myAgent.GLIE_policy(myAgent.s, T)
        myAgent.next_s = myMDP.next_state(myAgent.s, myAgent.a)
        myAgent.Q[s,a] = myAgent.Q[s,a] + lr *
        (myMDP.R[s] + beta * max(myAgent.Q[myAgent.next_s[0]]) - myAgent.Q[s,a])

    policy = np.argmax(myAgent.Q, axis=1)
    myAgent.set_policy(policy) # greedy policy for evaluation
    return myAgent

def reinforcement_learning(myMDPs, learning_num, max_trial, max_steps):
    for myMDP in myMDPs:
        myAgent = agent.Agent(myMDP)
        myAgent.init_state(myMDP.init_state())
        for i in range(2,learning_num):
            for j in range(0, max_trial):
                myAgent = Q_learning(myMDP, myAgent, max_steps, i)
            avg_reward = evaluation(myMDP, myAgent, max_trial, max_steps)

def main():
    """ PART 3 """
    learning_num = 500
    max_trial = 10
    reinforcement_learning(myMDPs, learning_num, max_trial, max_steps)
```

- **Performance of Q-learning**

Throughout 500 times learning, the agent converges to a certain point of high average rewards showing improvement of it behavior in the parking environment (Figure 1 and 2) Although the rewards of the learning are close to the reward from my polices, the converged rewards is not as good as the optimal policy I designed for each MDP. However, it generates the better performance than both random policies. I guess that the reason that the learned performance is not as good as the best policy is because the agent needs to do more explore to capture the optimal solution. After repeating more trials, I expect the learned performance gets closer to the optimal solution.

Policy	myMDP1	myMDP2
Random policy	-13.7980	-97.1944
Partial random policy	24.1406	-37.2679
My policy 1	32.0883	-43.1916
My policy 2	26.6502	-14.4402
Q learning	25.3379	-20.1584

Table 2: The average reward from Q-learning in each MDPs

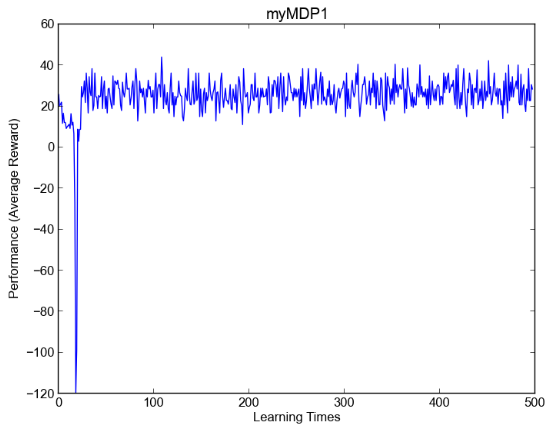

Figure 1: The performance for myMDP1

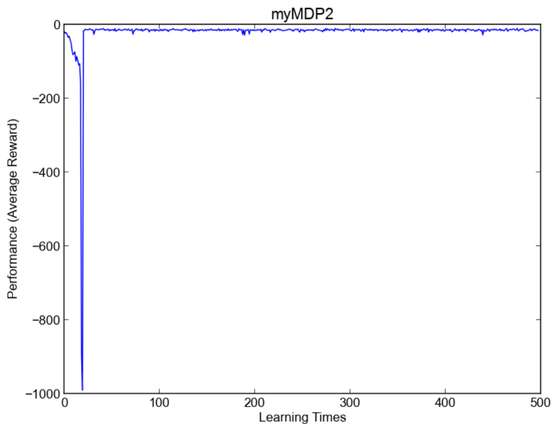

Figure 2: The performance for myMDP2