CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2020)– Lawson Wong

Name: [Virender Singh]
Collaborators: [None]

Code Execution To check the working of the code please execute main.py, it will ask you to choose the question number to run, you can enter question number to execute the corresponding script and check the submission.

Question 1. *Simulation function that acts as the four room environment. It should take in the current state s and desired action a, and return the next state s' as well as the reward r.*

Response:
Start state : (0, 0), Target state : (10, 10)
The code for the function can be found in the assignment in Simulation.py named as transition. The function takes as input the current state and the action, it then generates a random number between 0 and 1. If the number generated is below 0.8, it moves in the given direction, else it can go in any of the two perpendicular direction. If the next state is out of bound or hits a roadblock the current state remains the same and if it is the target, then the agent gets a reward of 1. The resulting state and the reward is returned to the agent. The python implementation is shown at figure 1 and can be found in the file Simulation.py of the project. Here the lookup tale generates a reward of 0 for all other policies except learned policy.

```
def transition(self, current_state, action):
    print("simulating")
    decision_prob = random()
    print("decision probability : ", decision_prob)
    if decision_prob < self.grid.prob:
        current_state.move(action.action_list[action.action])
    elif self.grid.prob < decision_prob < self.grid.prob + (1 - self.grid.prob) / 2:
        current_state.move(action.get_left_perpendicular())
    else:
        current_state.move(action.get_right_perpendicular())
    reward = 0
    print("current state : ", current_state.x, " ", current_state.y)
    if self.grid.is_target(current_state):
        current_state.__setstate__(self.grid.source[0], self.grid.source[1])
        reward = 1
    else:
        reward = self.policy.lookup[current_state.x][current_state.y][action.action]
    return current_state, reward
```

Figure 1: Function to simulate transition from one state to another. Takes as input current state and action and outputs the resulting state and reward.

Question 2. *Implement a manual policy, as well as an agent that interacts with the simulator and the policy.*

Response:
The manual policy asks the user for input action and returns the same. The policy is then used by simulate function in Simulation.py to execute transition with the help of transition function. The *get_action* is the main function of this class which outputs the action. The implementation is shown at figure 2:

```python
class ManualPolicy:
    def __init__(self):
        self.name = "Manual"
        self.lookup = [[{"up": 0, "down": 0, "left": 0, "right": 0}] * 11] * 11
        return

    def get_action(self, curr_state):
        print("input action")
        action_input = input()
        return action_input
```

Figure 2: Manual Policy class with get action function which takes user inputs and sends corresponding action to the user.

Question 3. *Implement a random policy. A random policy outputs one of the four actions, uniformly at random.*

Response:
A random policy just outputs any of the four actions when called by the simulate function. The available actions are "Up", "Down","Left", "Right". The *get_action* is the main function of this class which outputs the action. I have run random-policy agent for a total of 10 trials, 10^4 steps

```python
import random

class RandomPolicy:
    def __init__(self):
        self.name = "Random"
        self.lookup = [[{"up": 0, "down": 0, "left": 0, "right": 0}] * 11] * 11

    def get_action(self, curr_state):
        return random.choice(["up", "down", "left", "right"])

    def print_lookup(self):
        for i in range(len(self.lookup)):
            for j in range(len(self.lookup[0])):
                print(i, " ", j, " ", self.lookup[i][j])
```

Figure 3: Random Policy class with get action function which outputs any of the four available actions.

in each trial, and produced a cumulative reward plot as shown at figure 4. The dotted lines are the cumulative reward curves for each of 10 trials, and the thick solid black line is the average of

those 10 dotted curves.

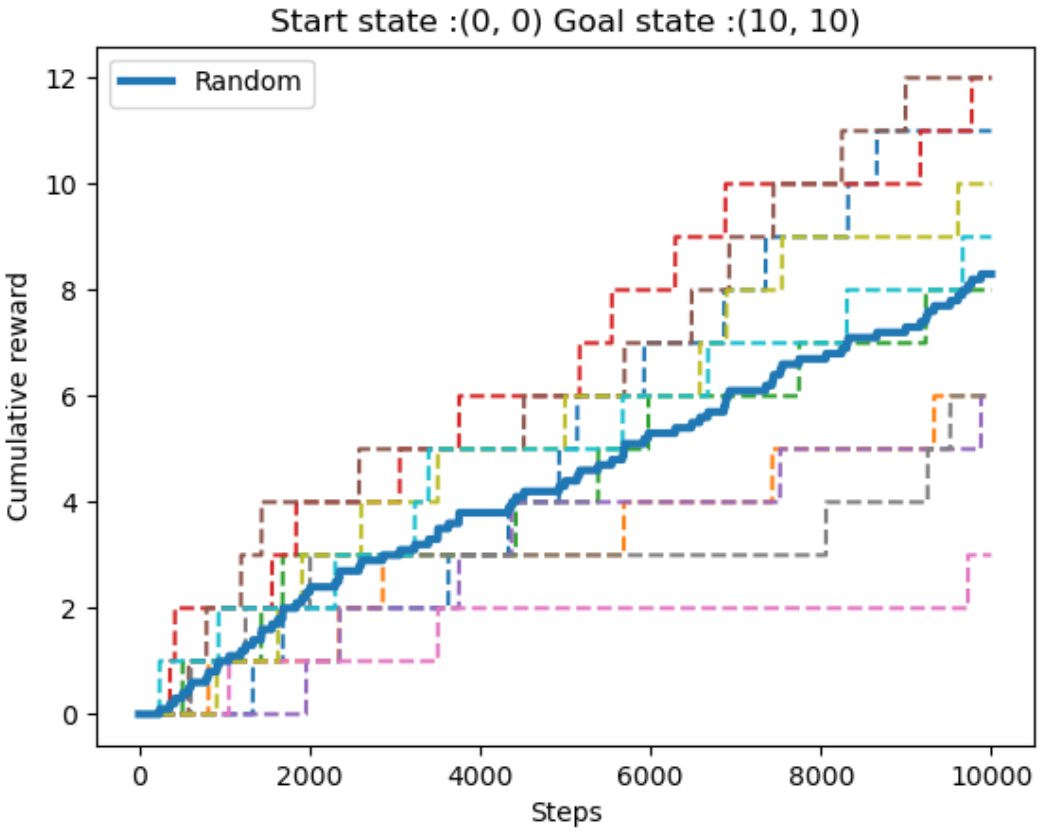

Figure 4: Cumulative reward plot for the random policy on 10 trials and $10^4 steps$.

Question 4. *Devise and implement at least two more policies, one of which should be generally worse than the random policy, and one better. Describe the overall strategy each of these policies uses, and why that leads to generally worse/better performance. Show the cumulative reward curves of each, similar to Q3.*

Response:
Worse Policy: The worse policy I have devised is a Go Left policy basically for each state the agent gives the action that go to left. This is a worse policy because the goal exists at the top right corner position i.e. (10, 10) and with this policy the agent will always go left and hence won't be able to reach target at any point of time. The policy is shown in figure 5 code snippet :

```
class WorsePolicy:
    def __init__(self):
        self.name = "Worse"
        self.lookup = [[{"up": 0, "down": 0, "left": 0, "right": 0}] * 11] * 11
        return

    def get_action(self, curr_state):
        return "left"
```

Figure 5: The worse policy returns left action for any state.

Better Policy: The better policy makes use of the location of the target, it checks the relative position of the current state w.r.t. target state. Taking the target state as the origin, we can locate the current state lying in one of the quadrants of this plane. Then the following cases can occur.

- If current state lies in the first quadrant i.e. current state is in north-east of the target state then it returns actions "left" and "down" with equal probability.
- If current state lies in the second quadrant i.e. current state is in north-west of the target state then it returns actions "right" and "down" with equal probability.
- If current state lies in the third quadrant i.e. current state is in south-west of the target state then it returns actions "right" and "up" with equal probability.
- If current state lies in the fourth quadrant i.e. current state is in south-east of the target state then it returns actions "left" and "up" with equal probability.

I have added some randomness in the better policy as well, so that the agent explores other areas as well and not just runs strictly according to the policy. This policy will always work better than the random policy because it is using the information of the target location and takes the step towards the target greedily. Therefore, every action returned by the policy takes the agent closer to the target. The policy is shown in code snippet in figure 6.
I have run all three agents, the random-policy agent, the worse policy agent and the better policy

```python
class BetterPolicy:
    def __init__(self, target):
        self.name = "Better"
        self.lookup = [[{"up": 0, "down": 0, "left": 0, "right": 0}] * 11] * 11
        self.target = target
        return

    def get_action(self, curr_state):
        action_prob = random()
        if action_prob < 0.7:
            return choice(["up", "down", "left", "right"])
        if curr_state.x<self.target[0]:
            if curr_state.y<self.target[1]:
                return choice(["up", "right"])
            else:
                return choice(["down", "right"])
        else:
            if curr_state.y<self.target[1]:
                return choice(["up", "left"])
            else:
                return choice(["down", "left"])
        return choice(["up", "down", "left", "right"])
```

Figure 6: The better policy always takes a step towards the goal. In that sense it is better than the random policy because it takes into consideration actions which are more likely to make it reach the target.

agent for a total of 10 trials, 10^4 steps in each trial, and produced a cumulative reward plot as shown below. The dotted lines are the cumulative reward curves for each of 10 trials, and the thick solid black line is the average of those 10 dotted curves. We can see in the graph that the better policy is performing better than the random policy while the worse policy is the getting least reward i.e. 0 which means it never reaches the target. The better policy is getting highest cumulative reward due to reaching the target most number of times. The plot is shown in figure 7.

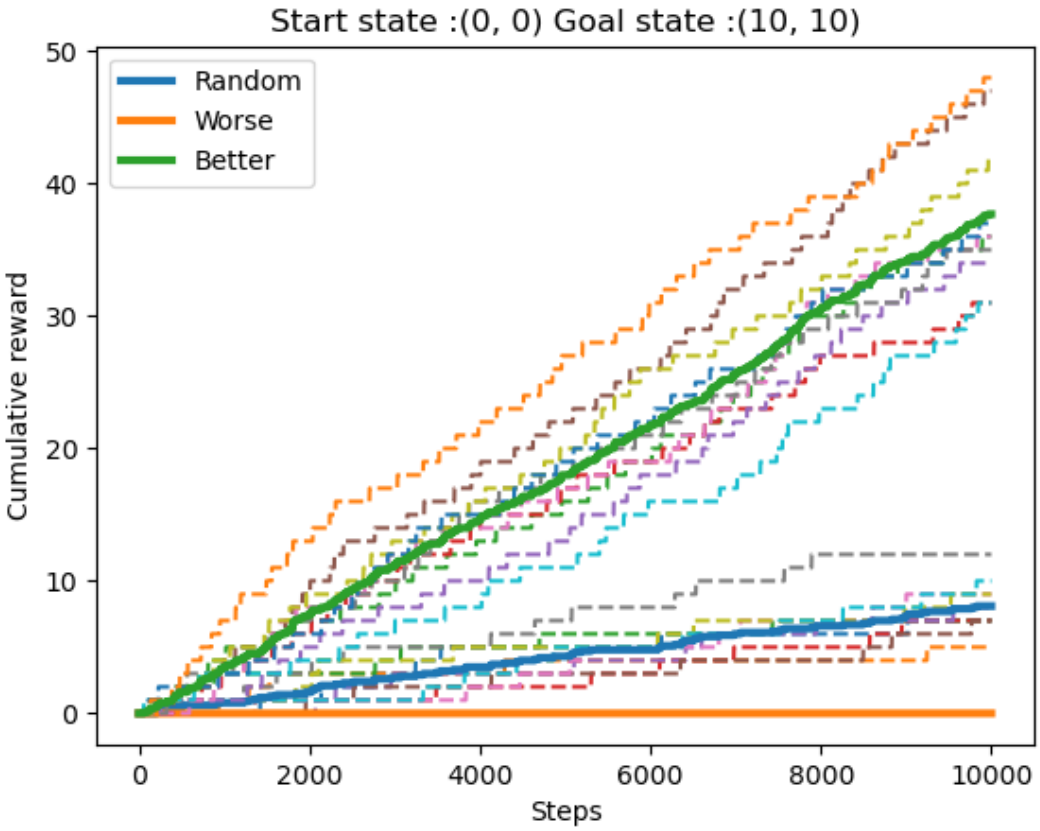

Figure 7: Cumulative reward plot for the random policy, better policy and the worse policy on 10 trials and 10^4 steps. The better policy is performing the best followed by random followed by the worse policy.

Question 5. *Devise and implement a learning agent. To facilitate this, first modify your code to select a random goal state before any trial, instead of using the fixed goal state (10, 10). The goal state should remain fixed throughout all trials. Your agent should learn where to find high reward and use this knowledge to act better in the future. Make sure your agent does not "cheat" by seeing what the goal state is – only the simulator can access this.*

Response:

Learned Policy : This policy doesn't have access to the target state. It can't use such information and acts solely basis on the rewards it achieves and tries to move in the direction of the higher reward. For this, we use a lookup table which given a state and action values can tell us which action to take. When we take a particular action in a particular state, we update its value in the lookup table with the reward we get from the next state using the transition function. If the action value for a particular state action pair is x and the reward we get is higher than this i.e. $r > x$, then we update the value for this state, action pair using this new reward. The transition function is the same we used in the first question and it tells us the next state and the reward

for any state and action pair. When asked for an action at a particular state, the policy looks up the actions with the highest values and if there are more than one such actions, takes decision randomly and if there is only one, it returns that action. The algorithm is described in below Pseudocode.

Algorithm 1: Learned Policy

```
Input : State
Output : Action
Data: Lookup action value table L(s,a), ϵ
x← random(0,1)
If x < ϵ :
    return random action
Else
    curr_state_actions = lookup all the available actions
    action_list = list of best actions according to lookup table
    best_action = choose random action out of best action list
```

The lookup table of learned policy is updated according to the following equation:

$$nextstate, reward = Transition(currentstate, action)$$
$$L(s,a) = max(L(s,a), reward)$$

where reward is obtained from the transition function and is used to update the corresponding entry in the lookup table.
The code snippet is shown in figure 8: I have run all four agents, the learned policy agent, the random-policy agent, the worse policy agent and the better policy agent for a total of 10 trials, 10^4 steps in each trial, and produced a cumulative reward plot as shown in figure 12. The only difference here being the target state is assigned randomly here and not known to the agent before hand. The dotted lines are the cumulative reward curves for each of 10 trials, and the thick solid black line is the average of those 10 dotted curves. We can see in the graph that the learned policy is performing the best and the better policy is performing better than the random policy while the worse policy is the getting least reward i.e. 0 which means it never reaches the target. The learned policy is getting highest cumulative reward due to reaching the target most number of times. We can see in the the figure 12 that it gets the highest reward for any goal state.

```python
class LearnedPolicy:
    def __init__(self):
        self.name = "Learned"
        self.lookup = [[{"up": 0, "down": 0, "left": 0, "right": 0}] * 11] * 11
        self.epsilon = 0.1

    def get_action(self, curr_state):
        action_prob = random.random()
        if action_prob > self.epsilon:
            curr_state_actions = self.lookup[curr_state.x][curr_state.y]
            all_values = curr_state_actions.values()
            max_value = max(all_values)
            action_list = []
            for i in curr_state_actions:
                if curr_state_actions[i] == max_value:
                    action_list.append(i)
            best_action = random.choice(action_list)
            print("best action according to learned policy : ", best_action)
            return best_action
        return random.choice(["up", "down", "left", "right"])
```

Figure 8: Learned Policy get action method returns the action with highest value

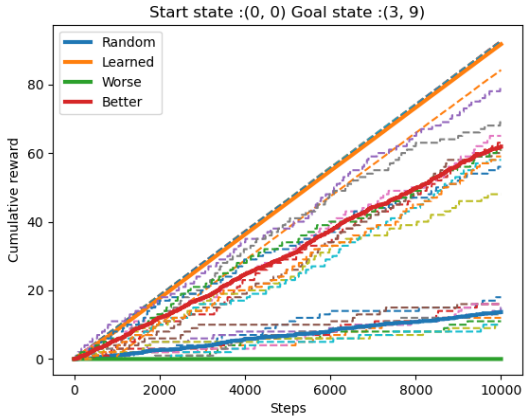

Figure 9: Goal state (3,9)

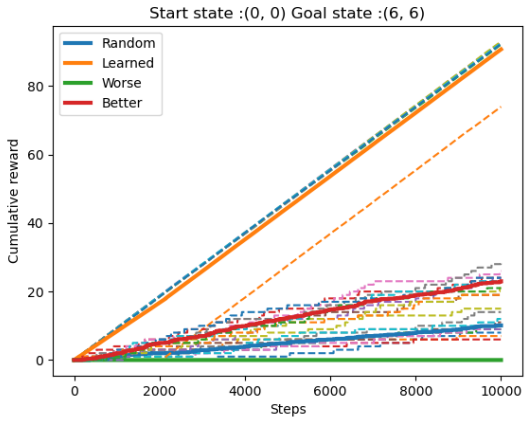

Figure 10: Goal state (6,6)

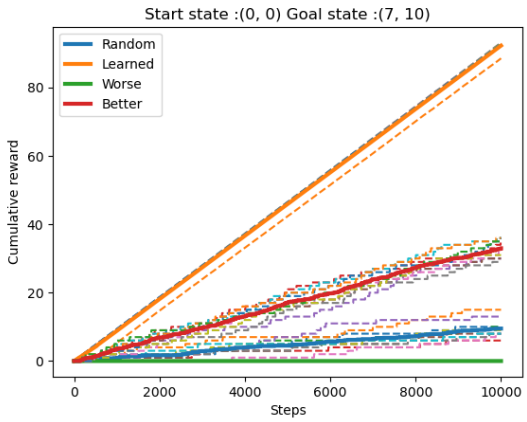

Figure 11: Goal state (7,10)

Figure 12: The figure shows performance of all four policies in different goal states. The goal states are generated randomly before any trial and is same for all ten trials.