ECEN 524 HW4
Aly Khater

Notebook for this homework can be found here:
https://colab.research.google.com/drive/1RirIfEcaF76VDyRkUQ1oXYdEYwBwGM0A?usp=sharing

Problem 1

POMDP vs MDP
POMDP utilizes a model-based approach by applying uncertainties in its training, unlike MDP. POMDP has the ability to to get new information based off of noisy observations based off of past states and actions. This is more robust for a robotic system as sensors are not ideal and typically have noisy observations, do not move perfectly, and can react to disturbances in the environment. As said before, the main differences between POMDP and MDP is the ability to model uncertainties such as in the following applications:

- Localization and Navigation
    - Robot location, environment map
- Autonomous Driving
    - Locations of traffic agents, control effects, other human driver's behaviours, weather, occlusion.
- Search and Tracking
    - Target location, target behavior
- Manipulation
    - Object pose, grasp success/failure, visual ambiguity, occlusion
- Human-Robot Interaction
    - Human intention
- Multi-Robot Coordination
    - Information private to each robot

Compared to DMPs, POMDMPs also have challenges, especially pertaining to the model uncertainty. Obtaining the information required to model these uncertainties is challenging, as states and actions can go unsolved. If a robot has not experienced anything related to uncertainty, how can it predict it? POMDP approaches also may require simplifications of the model.

Problem 2

Acknowledging ChatGPT and ECEN 522:Reinforcement Learning, utilizing my previous code (from scratch) and augmenting it.

1) Describe the robotic task and show a diagram of the MDP (similar to the example in the lecture)

The robotic task is to navigate through a 2D plane to reach the goal. I will call this robot a Roomba. The diagram of the Markov Decision Processes(MDP) is shown below in Figure 1.

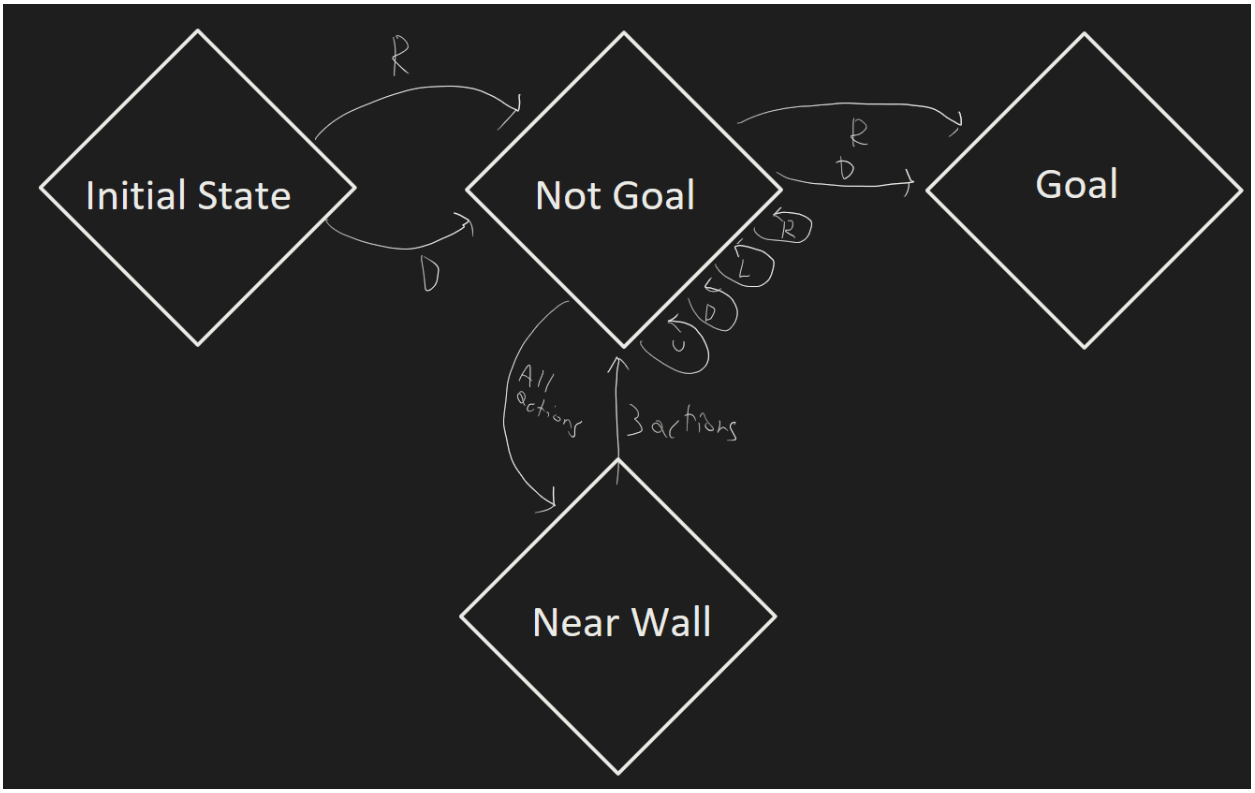

Figure 1: MDP diagram showcasing actions according to its position

I am utilizing a 5x5 GridWorld with obstacles placed in the gridworld, simulating an environment for the robot to navigate through as shown in Figure 2.

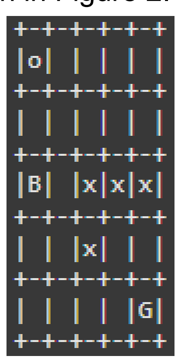

Figure 2: Gridworld layout

In the GridWorld, the *o* shows the current position of the robot, *x* defines the obstacle or wall the robot must avoid,*B* represents a termination state that immediately ends the program, failing the task, and *G* represents the goal state. The code to create the gridworld can be found below:

```python
def GridWorld5x5(p=0.9):
    rewards = {
        (2,0): -100,
        (4,4):  100
    }
    walls = [(2,2), (2,3), (2,4), (3,2)]

    # deterministic transition table
    T = {
        (0,0): { 'R': (0,1), 'D': (1,0) },
        (0,1): { 'R': (0,2), 'L': (0,0), 'D': (1,1) },
        (0,2): { 'R': (0,3), 'L': (0,1), 'D': (1,2) },
        (0,3): { 'R': (0,4), 'L': (0,2), 'D': (1,3) },
        (0,4): { 'L': (0,3), 'D': (1,4) },

        (1,0): { 'R': (1,1), 'D': (2,0), 'U': (0,0) },
        (1,1): { 'R': (1,2), 'L': (1,0), 'D': (2,1), 'U': (0,1) },
        (1,2): { 'R': (1,3), 'L': (1,1), 'U': (0,2) },
        (1,3): { 'R': (1,4), 'L': (1,2), 'U': (0,3) },
        (1,4): { 'L': (1,3), 'U': (0,4) },

        (2,0): { 'R': (2,1), 'D': (3,0), 'U': (1,0) },
        (2,1): { 'L': (2,0), 'D': (3,1), 'U': (1,1) },

        (3,0): { 'R': (3,1), 'D': (4,0), 'U': (2,0) },
        (3,1): { 'L': (3,0), 'D': (4,1), 'U': (2,1) },
        (3,3): { 'R': (3,4), 'D': (4,3) },
        (3,4): { 'L': (3,3), 'D': (4,4) },

        (4,0): { 'R': (4,1), 'U': (3,0) },
        (4,1): { 'R': (4,2), 'L': (4,0), 'U': (3,1) },
        (4,2): { 'R': (4,3), 'L': (4,1) },
        (4,3): { 'R': (4,4), 'L': (4,2), 'U': (3,3) },
        (4,4): { 'L': (4,3), 'U': (3,4) },
    }

    for s in T:
```

```
        ns_l = list(T[s].values())
        for a in T[s]:
            ns = T[s][a] #next state
            rs = ns_l[np.random.choice(len(ns_l))] #random
            if ns == rs:
                T[s][a] = { ns: 1.0 }
            else:
                T[s][a] = { ns: p, rs: np.round(1-p,2) }

    g = GridWorld(5, 5, start_position=(0, 0),
            pass_through_reward=0, rewards=rewards, walls = walls)
    g.probs = T

    return g
```
where T allocates the available actions at the current state. I assign *G* with a +100 reward to encourage reaching the goal state, while the termination state has a -100 reward. I also incorporate some reward changes between the goal and initial state, to discourage oscillations and encourage actually reaching the goal. Without having these intermediate rewards, the MDP would sometimes never reach the goal state as it learns that oscillations in place would maximize the reward by avoiding the termination zone.

The Goal state is defined as (4,4) in the gridworld, and the termination state is defined as (2,0). These can be seen in Figure 4.

```
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| |o|x| | |
+-+-+-+-+-+
| | | | |G|
+-+-+-+-+-+
```

Figure 4: Another example of Gridworld, with the robot in a different state.

Each block in the Gridworld that the robot can navigate is a state, with a total of 21 states available to the robot as it cannot be on a wall. There are a total of 4 actions the robot can perform: “Up”, “Down”, “Left”, “Right.”

2) Copy the code you implemented for the MDP and for the value iteration.

```
from gridworld_hw3_q1 import GridWorld
#from gw55_hw3 import *
import numpy as np
import os
import random

# reward shaping global constants
REWARD_BONUS = 4           # Bonus for moving closer to the goal
PENALTY = -1               # Penalty for moving further away
OSCILLATION_PENALTY = -6   # Penalty for oscillatory moves
GOAL_STATE = (4, 4)

def manhattan_distance(state, goal):
    #distance from current position to goal
    return abs(state[0] - goal[0]) + abs(state[1] - goal[1])

def get_states(g):
    augmented = []
    for s in g.all_states():
        # Initial augmented state when no previous state exists
        augmented.append((s, None))
        for s_prev in g.all_states():
            augmented.append((s, s_prev))
    return augmented

def value_iteration(g, gamma=0.9, theta=1e-4, max_iterations=1000):
    augmented_states = get_states(g)
    # Initialize value function
    V = {state: 0 for state in augmented_states}
    iteration = 0

    while True:
        delta = 0
        iteration += 1
        for (s, last) in augmented_states:
            # If s is terminal, skip update
            if s in g.rewards:
```

```
                continue

            best_value = float('-inf')
            possible_actions = g.actions(s)

            # V(s) equation for each action
            for a in possible_actions:
                value = 0
                next_states_probs = g.probs.get(s, {}).get(a, {})
                for s_next, prob in next_states_probs.items():

                    new_state = (s_next, s)
                    base_reward = g.world[s_next]

                    # Apply oscillation penalty if s_next equals the last
state
                    extra_penalty = OSCILLATION_PENALTY if (last is not
None and s_next == last) else 0
                    r = base_reward + extra_penalty
                    value += prob * (r + gamma * V.get(new_state, 0))
                best_value = max(best_value, value)
            delta = max(delta, abs(best_value - V[(s, last)]))
            V[(s, last)] = best_value
        if delta < theta or iteration >= max_iterations:
            break

    # Extract optimal policy based on augmented states and value function
    policy = {}
    for (s, last) in augmented_states:
        if s in g.rewards:
            continue      #Skip our terminal states
        best_action = None
        best_value = float('-inf')
        possible_actions = g.actions(s)
        for a in possible_actions:
            value = 0
            next_states_probs = g.probs.get(s, {}).get(a, {})

            # Applies the oscillation penalty to discourage staying in the
same place
```

```
            for s_next, prob in next_states_probs.items():
                new_state = (s_next, s)
                base_reward = g.world[s_next]
                extra_penalty = OSCILLATION_PENALTY if (last is not None
and s_next == last) else 0
                r = base_reward + extra_penalty
                value += prob * (r + gamma * V.get(new_state, 0))
            if value > best_value:
                best_value = value
                best_action = a
        policy[(s, last)] = best_action

    return policy, V
```

Code to run the gridworld

```
if __name__ == '__main__':
    g = GridWorld5x5()
    policy, V = value_iteration(g,gamma=0.9)

    # Initialize augmented state: starting at g.start_position with no
last state
    current_augmented_state = (g.start_position, None)

    while not g.game_over():
        g.print()
        print()

        current_state, last_state = current_augmented_state
        actions = g.actions(current_state)
        # Choose action from policy for the augmented state
        action = policy.get(current_augmented_state,
random.choice(actions))
        print(f"Chosen action: {action}")

        next_state, reward = g.move(action)

        # Compute Manhattan distances for reward shaping
        md_current = manhattan_distance(current_state, GOAL_STATE)
        md_next = manhattan_distance(next_state, GOAL_STATE)
        if md_next < md_current:
```

```
            reward += REWARD_BONUS
            print(f"Reward bonus for moving closer: {REWARD_BONUS}")
        elif md_next > md_current:
            reward += PENALTY
            print(f"Penalty for moving further: {PENALTY}")


        print(f"Moved to state: {next_state}")
        print(f"Reward: {reward}")

        # Update the agent's state
        g.set_state(next_state)
        current_augmented_state = (next_state, current_state)

    # Print the optimal augmented policy and value function
    print("Optimal Policy:")
    for r in range(g.rows):
        for c in range(g.columns):
            key = ((r, c), None)
            if key in policy:
                print(f"{(r, c)}: {policy[key]}", end=", ")
        print()

    print("Final Value Function:")
    for r in range(g.rows):
        for c in range(g.columns):
            key = ((r, c), None)
            print(f"{(r, c)}: {V.get(key, 0):.2f}", end=", ")
        print()
```

3) Provide screenshots from your computer show that you run the code, including showing the results. Your code should output the optimal path.

In Figure 5, this is the output where I ran with a discount factor (gamma) of 0.9.

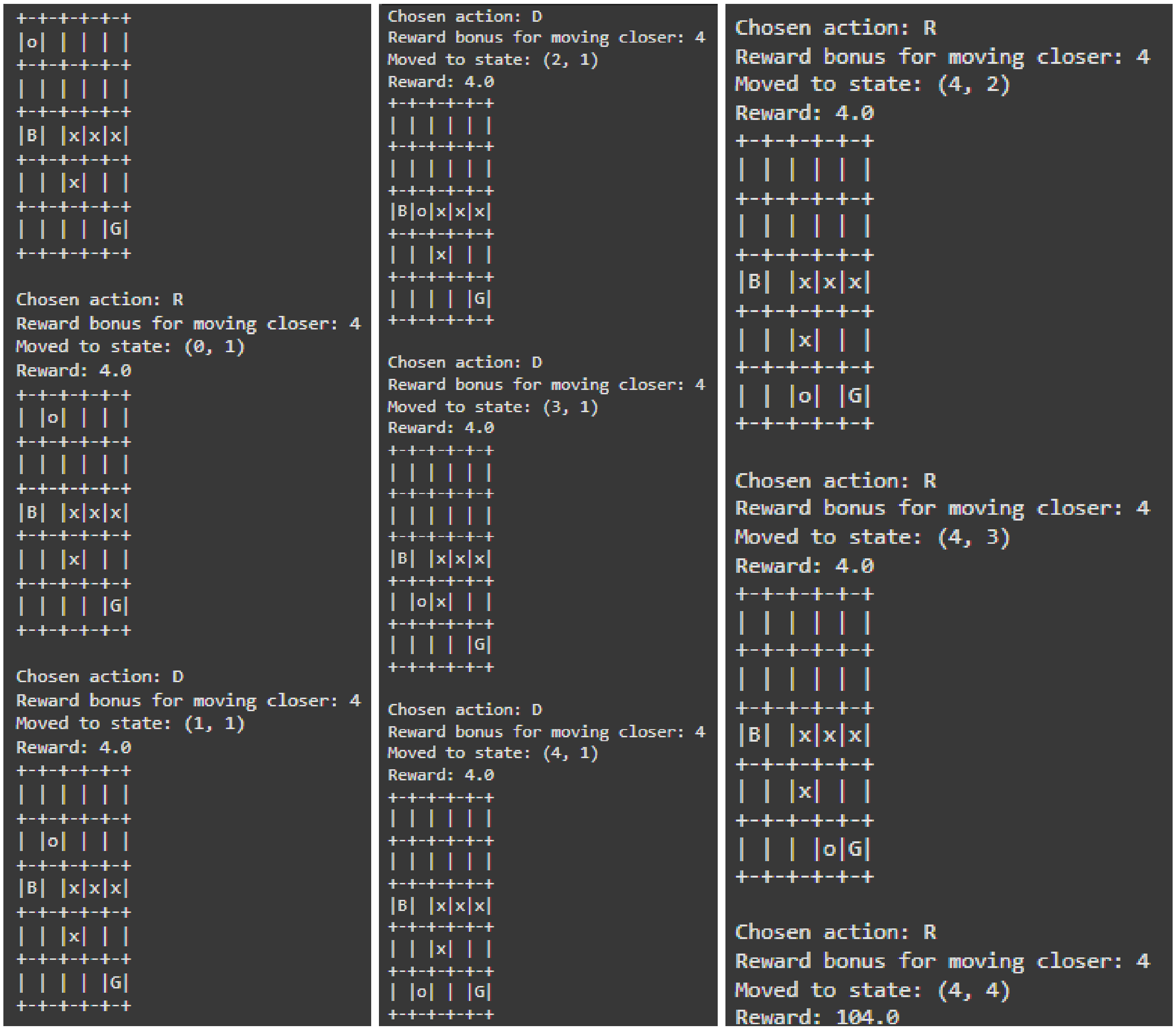

Figure 5: Output of the gridworld, from top to bottom, left to right. Gamma = 0.9

As you can see, after training, the robot finds the optimal path to the goal state and gets a positive reward at each step. The optimal policy and optimal value function is shown in Figure 6:

```
Optimal Policy:
(0, 0): R, (0, 1): D, (0, 2): D, (0, 3): L, (0, 4): D,
(1, 0): R, (1, 1): D, (1, 2): L, (1, 3): L, (1, 4): L,
(2, 1): D,
(3, 0): R, (3, 1): D, (3, 3): D, (3, 4): D,
(4, 0): R, (4, 1): R, (4, 2): R, (4, 3): R,
Final Value Function:
(0, 0): 30.67, (0, 1): 34.09, (0, 2): 30.59, (0, 3): 26.93, (0, 4): 23.91,
(1, 0): 34.58, (1, 1): 39.35, (1, 2): 34.05, (1, 3): 29.52, (1, 4): 26.56,
(2, 0): 0.00, (2, 1): 45.40, (2, 2): 0.00, (2, 3): 0.00, (2, 4): 0.00,
(3, 0): 61.01, (3, 1): 68.39, (3, 2): 0.00, (3, 3): 87.56, (3, 4): 97.88,
(4, 0): 70.49, (4, 1): 78.32, (4, 2): 87.02, (4, 3): 97.29, (4, 4): 0.00,
```

Figure 6: Output of Optimal Policy and Final Value Function with Gamma = 0.9

The optimal value function gets much higher as the robot gets closer to the goal state.

4) Try with two different γ values (0.9 and 0.3) and show if there is a difference in your results. Explain why there is or there isn't a difference

Discount factor of 0.9 was shown in Problem 3. Figure 7 shows the output gridworld for a discount factor of 0.3.

```
+-+-+-+-+-+
|o| | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| | | | |G|
+-+-+-+-+-+

Chosen action: D
Reward bonus for moving closer: 4
Moved to state: (1, 0)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|o| | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| | | | |G|
+-+-+-+-+-+

Chosen action: R
Reward bonus for moving closer: 4
Moved to state: (1, 1)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| |o| | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| | | | |G|
+-+-+-+-+-+

Chosen action: D
Reward bonus for moving closer: 4
Moved to state: (2, 1)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B|o|x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| | | | |G|
+-+-+-+-+-+

Chosen action: D
Reward bonus for moving closer: 4
Moved to state: (3, 1)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| |o|x| | |
+-+-+-+-+-+
| | | | |G|
+-+-+-+-+-+

Chosen action: D
Reward bonus for moving closer: 4
Moved to state: (4, 1)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| |o| | |G|
+-+-+-+-+-+

Chosen action: R
Reward bonus for moving closer: 4
Moved to state: (4, 2)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| | |o| |G|
+-+-+-+-+-+

Chosen action: R
Reward bonus for moving closer: 4
Moved to state: (4, 3)
Reward: 4.0
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
| | | | | |
+-+-+-+-+-+
|B| |x|x|x|
+-+-+-+-+-+
| | |x| | |
+-+-+-+-+-+
| | | |o|G|
+-+-+-+-+-+

Chosen action: R
Reward bonus for moving closer: 4
Moved to state: (4, 4)
Reward: 104.0
```

Figure 7: Output of the gridworld, from top to bottom, left to right. Gamma = 0.3

The optimal policy and optimal value function is shown in Figure 8:

```
Optimal Policy:
(0, 0): D, (0, 1): D, (0, 2): L, (0, 3): L, (0, 4): L,
(1, 0): R, (1, 1): D, (1, 2): U, (1, 3): L, (1, 4): L,
(2, 1): D,
(3, 0): D, (3, 1): D, (3, 3): D, (3, 4): D,
(4, 0): R, (4, 1): R, (4, 2): R, (4, 3): R,
Final Value Function:
(0, 0): 0.01, (0, 1): 0.02, (0, 2): 0.00, (0, 3): -0.00, (0, 4): -0.02,
(1, 0): 0.04, (1, 1): 0.13, (1, 2): 0.00, (1, 3): 0.00, (1, 4): -0.02,
(2, 0): 0.00, (2, 1): 0.54, (2, 2): 0.00, (2, 3): 0.00, (2, 4): 0.00,
(3, 0): 0.40, (3, 1): 1.80, (3, 2): 0.00, (3, 3): 27.18, (3, 4): 90.80,
(4, 0): 1.96, (4, 1): 7.28, (4, 2): 27.01, (4, 3): 90.63, (4, 4): 0.00,
```

Figure 9: Output of Optimal Policy and Final Value Function with Gamma = 0.9

You can see a stark difference in the Final Value function between Gamma=0.9 (Figure 7) and Gamma=0.3 (Figure 9). Near the beginning of the GridWorld, its learned maximum reward at those states is 30 for Gamma=0.9, and nearly 0 for Gamma=0.3. The discount factor of 0.3 places a harsher penalty on rewards, so it finds that lingering near the beginning of the GridWorld does not provide much rewards. With the Gamma=0.9, the discount factor is much lighter, so it can explore but also make less optimal decisions while still getting a decent reward. There are even negative values as I have implemented a penalty reward for moving away from the goal state or oscillating. Due to the discount factor 0.3, these negative values are high at the beginning, then as it approaches the goal state, the reward is already discounted highly that reaching the goal state changes the reward negligibly.