Imperial College London

Department of Computing

Reinforcement Learning Coursework 1

Author:

Yong Shen Tan
(CID 01055349)

November 10, 2020

Question 1: Understanding of MDPs

Relevant Python code for this question can be found in Appendix A.

Part 1.a

The sequences of states and rewards computed using the CID of **1055349** produces the following trace:

$$\tau = s_3\ 1\ s_2\ 0\ s_3\ 1\ s_3\ 1\ s_1\ 3\ s_2\ 0\ s_3\ 1 \tag{1}$$

Part 1.b

From the observed trace above, we can construct the following MDP graph:

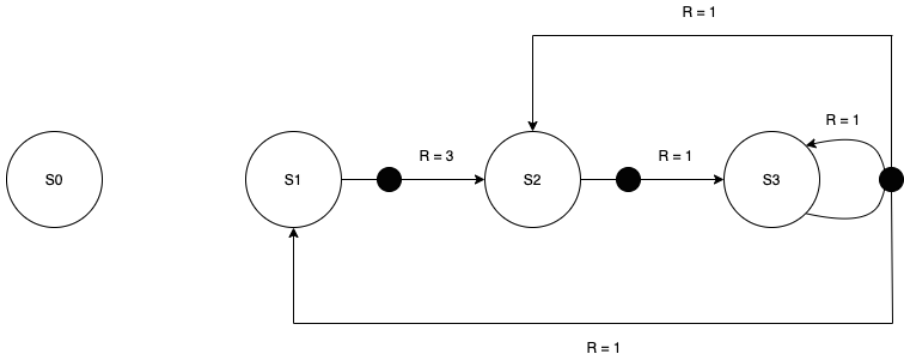

Figure 1: Minimal MDP Graph from Observed Trace

Following the trace, we can draw the transitions between states, where the action chosen is the deterministic one, *"no choice"*, represented as a black circle. The trace also gives us the immediate rewards in that time step as we transition between states, indicated by the arrows. We do the same for all state transitions in the trace, eventually obtaining the MDP graph seen in Figure 1. In the case where the trace goes from one state to itself, such as s_3 to s_3 as seen in the trace, we draw a self-connecting transition.

Part 1.c (1)

The transition probabilities are independent of the action we choose, allowing us to describe the transition matrix of the MDP as a 4x4 matrix, $P_{ss'}$:

$$P_{ss'} = \begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix} \tag{2}$$

$$P_{ss'} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix} \tag{3}$$

Based on the trace observed we can say for certain that the probability of transitioning between the states with arrows shown in Figure 1 is more than 0. The other transition probabilities are unknown and cannot be determined as we have not observed those state transitions. That is to say:

$$p_{ss'} \in [p_{12},\ p_{12},\ p_{31},\ p_{32},\ p_{33}] > 0 \tag{4}$$

$$p_{ss'} \notin [p_{12},\ p_{12},\ p_{31},\ p_{32},\ p_{33}] \in [0, 1] \tag{5}$$

Assuming that our single trace fully describes the transition dynamics of the MDP, we obtain the following transition matrix, $P_{ss'}$, shown in (3).

Part 1.c (2)

The immediate rewards are independent of the action we choose, allowing us to describe the reward matrix of the MDP as a 4x4 matrix, $R_{ss'}$:

$$R_{ss'} = \begin{pmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{pmatrix} \quad (6) \qquad R_{ss'} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \quad (7)$$

Based on the trace observed we can say for certain that the immediate reward collected when transitioning between the states with arrows shown in Figure 1 is more than 0. The other immediate rewards are unknown and cannot be determined as we have not observed those state transitions. That is to say:

$$k_{ss'} \in [k_{12},\ k_{12},\ k_{31},\ k_{32},\ k_{33}] > 0 \quad (8) \quad k_{ss'} \notin [k_{12},\ k_{12},\ k_{31},\ k_{32},\ k_{33}] \rightarrow \textit{unknown} \quad (9)$$

Assuming that our single trace fully describes the reward dynamics of the MDP, we obtain the following reward matrix, $R_{ss'}$, shown in (7).

Part 1.c (3)

Given that we only have a single trace of the MDP, which might also be an incomplete episode, we are not able to use Monte-Carlo methods. We also do not have information about the transition or reward matrix of the MDP and are thus not able to use Dynamic Programming methods. This leaves us with Temporal Difference methods for estimating the value of s_3.

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad (10)$$

Applying the TD(0) algorithm on the observed trace of states and rewards, where we do iterative updates of our estimated value functions based on (10), we are able to compute $V(s_3) = 1$. A learning rate, α, of 1 was used given that we only have a single trace. The discount factor, γ, was set to 1 to obtain the undiscounted value of s_3.

Question 2: Understanding of Grid Worlds

Relevant Python code for this question can be found in Appendix B.

Part 2.a

Using the last 3 digits of the CID of **1055349**, we obtain the following:

$$s_{reward} = s_2 \quad (11) \qquad p = 0.45 \quad (12) \qquad \gamma = 0.4 \quad (13)$$

Part 2.b (1-3)

The optimal value function and policy were computed using the Dynamic Programming value iteration algorithm, shown in Figures 2 and 3 respectively, with the stopping condition $\theta = 0.00001$. When determining the optimal policy for a given state in the case where there are multiple equally optimal actions, we assume we randomly pick one from those actions with equal probabilities.

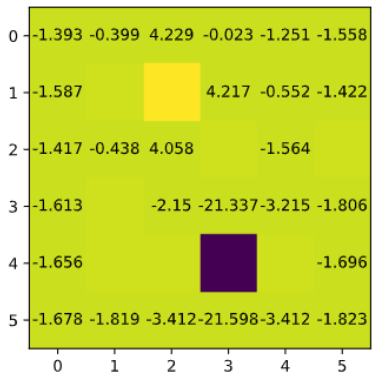

Figure 2: DP Optimal Value Function

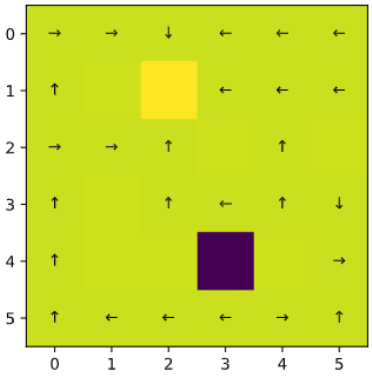

Figure 3: DP Optimal Policy

Part 2.b (4)

If we define the ideal policy as one that leads to the reward state in the least numbers of moves, increasing p improves the optimal policy towards the ideal policy; a higher p indicates there's less stochasticity in our agent's actions, allowing it to better estimate the value of states and hence a better policy. This was verified by testing with p values of 0.1, 0.25 and 0.9.

Increasing γ causes value iteration to converge more slowly to the optimal value function. A higher γ implies less discounting of rewards as the agent is more far-sighted, causing the difference of the estimated value function between iterations to be larger and thus requiring more iterations for the difference to fall below δ. This was verified by testing with γ values of 0.25 and 0.75.

Part 2.c (1-2)

The optimal value function and policy were computed using the Monte Carlo iterative optimisation algorithm, shown in Figures 4 and 5 respectively. The agent was trained over 1000 episodes using an ϵ-greedy policy, with $\epsilon = 0.1$, and a learning rate $\alpha = 0.05$. We assume that our agent would have explored each state at least $\frac{1}{\alpha}$ times, such that scaling by α is an accurate estimate of its value.

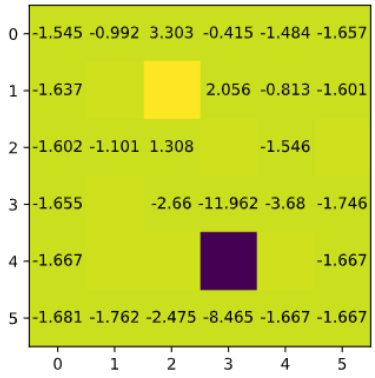

Figure 4: MC Estimate of Optimal Value Function

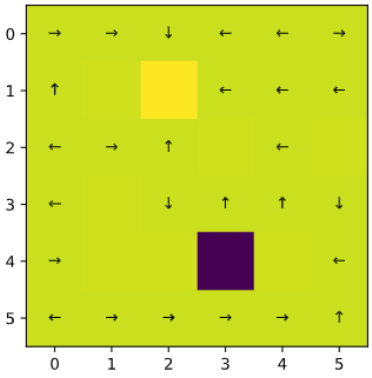

Figure 5: MC Estimate of Optimal Policy

Part 2.c (3)

The learning curve in Figure 6, derived using MC, was averaged over 1000 repetitions of 1000 episodes. 1000 repetitions was chosen as it sufficiently reduced noise between repetitions such that we can more clearly see the agent learning over episodes and the rewards reach a plateau.

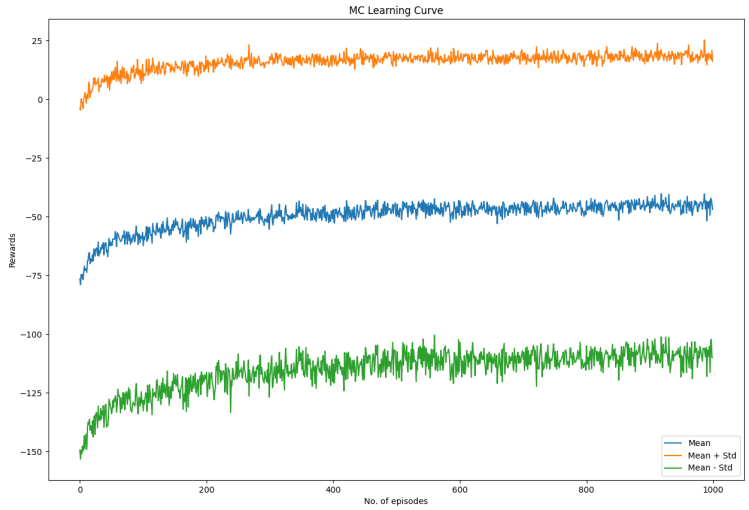

Figure 6: Learning Curve of MC Agent

Part 2.c (4)

Figure 7 shows that increasing α causes the agent to learn more quickly but plateau earlier. As the environment is stationary, giving more weight to recent episodes as our agent learns pulls the estimated value function faster to the optimal one. A lower α however, leads to more steady learning and higher eventual rewards, as more information is used to inform its policy.

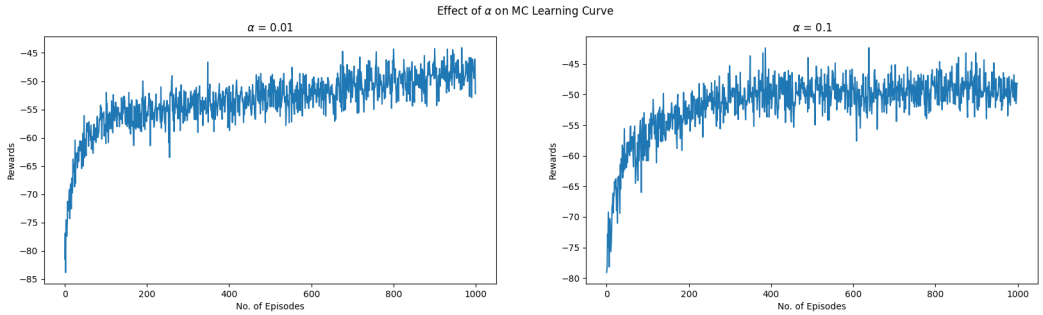

Figure 7: Learning Curve of MC Agent (Varying α)

Figure 8 shows that increasing ϵ causes slightly greater variance in the learning curve. This makes sense as our agent might encounter poor rewards while exploring the grid world more as it's not following the optimal policy.

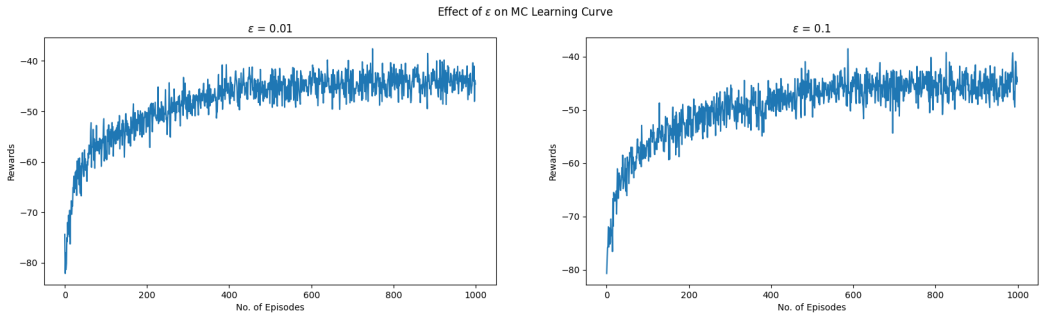

Figure 8: Learning Curve of MC Agent (Varying ϵ)

Part 2.d (1-2)

The optimal value function and policy were computed using the Q-Learning algorithm, shown in Figures 9 and 10 respectively. The agent was trained over 1000 episodes using an ϵ-greedy policy, with $\epsilon = 0.1$, and a learning rate $\alpha = 0.05$. We assume that our agent would have explored each state at least $\frac{1}{\alpha}$ times, such that scaling by α is an accurate estimate of its value.

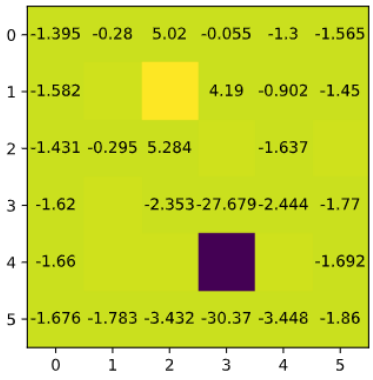

Figure 9: TD Estimate of Optimal Value Function

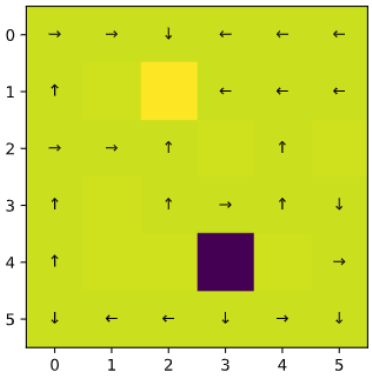

Figure 10: TD Estimate of Optimal Policy

Part 2.d (3)

The learning curve in Figure 11, derived using TD, was averaged over 1000 repetitions of 1000 episodes. 1000 repetitions was chosen as it sufficiently reduced noise between repetitions such that we can more clearly see the agent learning over episodes and the rewards reach a plateau.

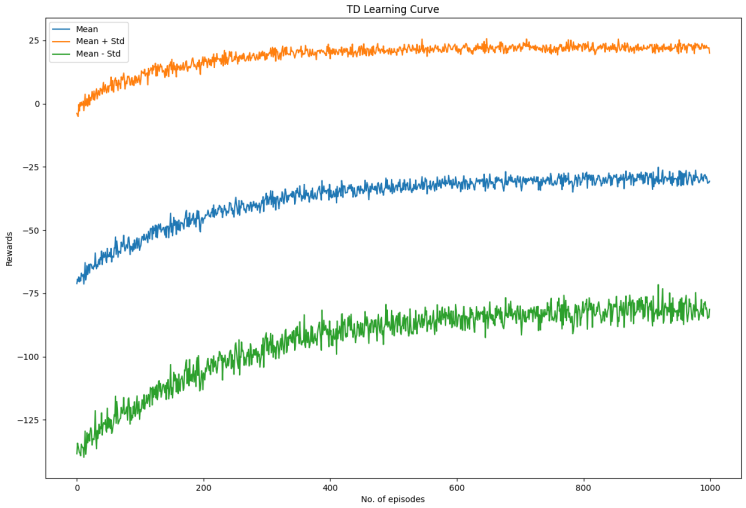

Figure 11: Learning Curve of TD Agent

Part 2.d (4)

Figure 12 shows that increasing α causes the agent to learn more quickly but plateau earlier. As the environment is stationary, giving more weight to recent episodes as our agent learns pulls the estimated value function faster to the optimal one.

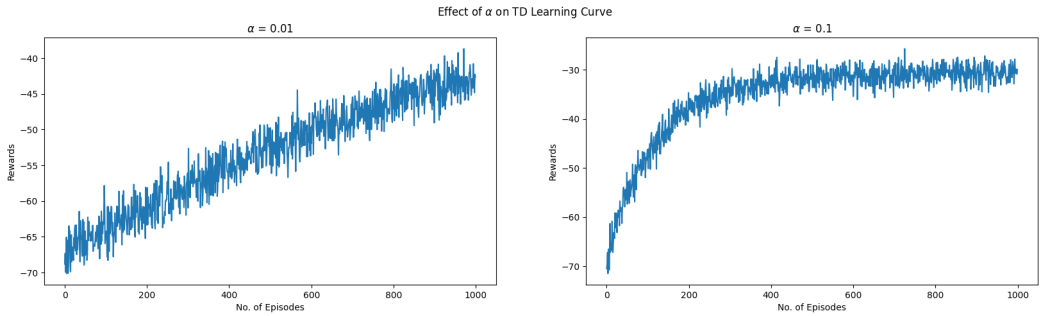

Figure 12: Learning Curve of TD Agent (Varying α)

Figure 13 shows that increasing ϵ causes slightly greater variance in the learning curve. This makes sense as our agent might encounter poor rewards while exploring the grid world more as it's not following the optimal policy.

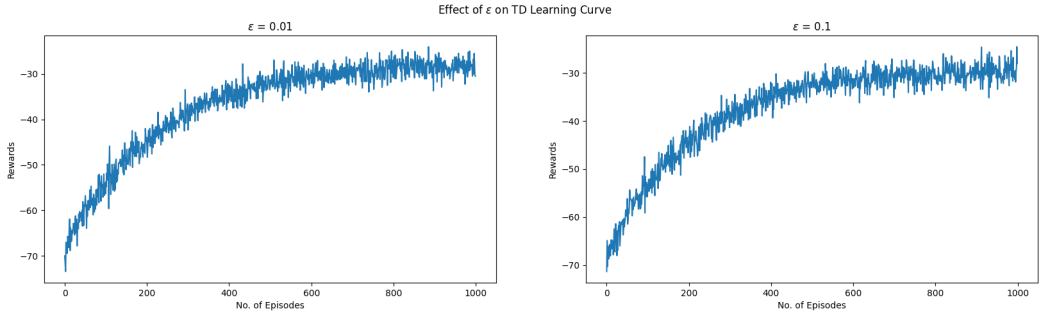

Figure 13: Learning Curve of TD Agent (Varying ϵ)

Part 2.e (1)

It is observed from Figure 14 that the TD method learns more quickly when estimating the optimal value function compared to the MC implementation, seen from the steeper slope. This agrees with the theory that TD methods are usually more efficient than MC methods at learning, as MC uses complete returns to improve estimates while TD immediately uses estimates of successor states.

Theoretically, MC methods should be more accurate compared to TD methods as they are less prone to bias. However, as we are randomly initialising our starting state each episode, this disadvantage of TD methods is reduced and it performs more accurately as seen from the lower estimation error compared to MC methods.

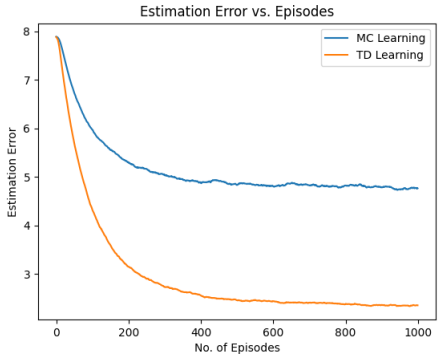

Figure 14: Estimation Error vs. Episodes

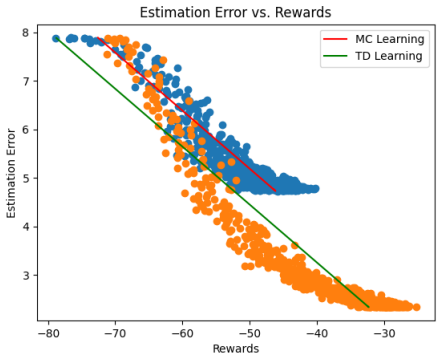

Figure 15: Estimation Error vs. Rewards

Part 2.e (2)

Figure 15 characterises how the rewards collected by the MC (blue) and TD (orange) agents during each episode depends on how accurately it estimated the optimal value function. The variability in the data accounts for the exploration done by our agents as well as the stochasticity of the environment.

The plots for both show a clear trend that as the estimation error decreases, our rewards improve as our agent is able to reap more rewards by choosing the most optimal action in each state that maximises its value. This highlights that it is highly important to have a good estimation of the optimal value function to gain higher rewards. However, a good estimate doesn't necessarily guarantee high rewards as there is high stochasticity in the environment, due to $p = 0.4$, that the agent cannot plan for.

Part 2.e (3)

As seen from Figure 14, the TD method outperforms the MC one at learning the optimal policy. Furthermore, in this case where exploring starts were used, the inherent bias of TD learning can be reduced such that it performs more accurately compared to MC learning. Figures 16 and 17 show that varying the learning parameters, ϵ and α, do not really change the performance of MC over TD.

However, as the grid world is a Markovian environment and TD methods exploit the Markov property, they tend to perform better in the given environment. It's possible for MC methods to outperform TD methods in a non-Markovian environment as MC is not dependent on the Markov property and instead uses empirical experiences.

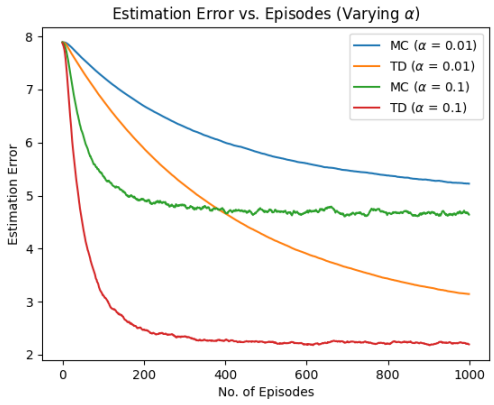

Figure 16: Estimation Error vs. Episodes With Varying α

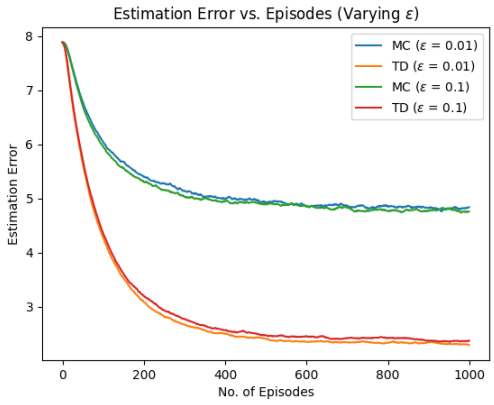

Figure 17: Estimation Error vs. Episodes With Varying ϵ

Appendices

A Python Code for Question 1

```
#!/usr/bin/env python
# coding: utf-8

CID = "1055349"

states = []
rewards = []

for digit in CID:
    states.append((int(digit) + 2) % 4)
    rewards.append(int(digit) % 4)

print("The observed trace of states and rewards:", end=" ")
for idx, value in enumerate(states):
    print(f"s{value}", end=" ")
    print(rewards[idx], end=" ")


V = [0, 0, 0, 0]
alpha = 1
gamma = 1

for i in range(len(states)-1):
    current = states[i]
    next = states[i+1]
    reward = rewards[i]
    V[current] += alpha * (reward + gamma * V[next] - V[current])

first = states[0]
print(f"The estimated value of the first state (s{first}) in the trace: {V[
    first]}")
```

B Python Code for Question 2

```
#!/usr/bin/env python
# coding: utf-8

import matplotlib.pyplot as plt
import numpy as np
from numpy import random


x, y, z = 3, 4, 9
reward_state = (z + 1) % 3 + 1
print(f"Reward state: s{reward_state}")
p = 0.25 + 0.5 * (x + 1) / 10
print(f"Value of p: {p}")
gamma = 0.2 + 0.5 * y / 10
print(f"Value of gamma: {gamma}")


class GridWorld(object):
    def __init__(self, success_prob):
```

```
        #### Attributes defining the grid world ####

        # Shape of the grid world
        self.shape = (6,6)

        # Locations of the obstacles
        self.obstacle_locs = [(1, 1), (2, 3), (2, 5), (3, 1), (4, 1), (4, 2)
            , (4, 4)]

        # Locations for the absorbing states
        self.absorbing_locs = [(1, 2), (4, 3)]

        # Corresponding rewards for each of the absorbing states
        self.special_rewards = [10, -100]

        # Reward for all the other states
        self.default_reward = -1

        # Initial location
        self.initial_loc = (3, 0)

        # Action names
        self.action_names = ["N", "E", "S", "W"]

        # Number of actions
        self.action_size = len(self.action_names)

        # Probabilities of the outcomes for each action
        self.action_prob = self.__generate_action_prob__(success_prob)

        ##########################

        #### Internal State ####

        # Get attributes defining the world
        state_size, T, R, absorbing, locs, neighbours = self.
            __build_grid_world__()

        # Number of valid states in the grid world (there are 29 of them)
        self.state_size = state_size

        # Transition operator (3D tensor)
        self.T = T

        # Reward function (3D tensor)
        self.R = R

        # Absorbing states
        self.absorbing = absorbing

        # Non-absorbing states
        self.non_absorbing = list(filter(lambda state: state not in self.
            absorbing, range(self.state_size)))

        # The locations of the valid states
        self.locs = locs

        # The neighbouring states in each direction for each state
        self.neighbours = neighbours
```

```
        # State number of the initial state
        self.initial_state = self.__loc_to_state__(locs, self.initial_loc);

        # Locating the initial state
        self.initial = np.zeros(len(locs))
        self.initial[self.initial_state] = 1

        # Placing the walls on a bitmap
        self.walls = np.zeros(self.shape);
        for ob in self.obstacle_locs:
            self.walls[ob] = 1

        # Placing the absorbers on a grid for illustration
        self.absorbers = np.zeros(self.shape)
        for ab in self.absorbing_locs:
            self.absorbers[ab] = -1

        # Placing the rewarders on a grid for illustration
        self.rewarders = np.zeros(self.shape)
        for i, reward in enumerate(self.absorbing_locs):
            self.rewarders[reward] = self.special_rewards[i]


    ############################

    #### Internal Helper Functions ####

    def __generate_action_prob__(self, success_prob):
        # Calculate the probability of each outcome that isn't the action we
            chose
        noise = (1 - success_prob) / (self.action_size - 1)

        # Fill the action-outcome matrix with noise except along the
            diagonal, which should be
        # our probability of successfully performing the action we chose
        action_prob = np.full((self.action_size, self.action_size), noise)
        np.fill_diagonal(action_prob, success_prob)

        return action_prob


    def __build_grid_world__(self):
        # Get the locations of all the valid states, the neighbours of each
            state (by state number),
        # and the absorbing states (array of bools with True in the
            absorbing states)
        locations, neighbours, absorbing = self.__get_topology__()

        # Get the number of states
        S = len(locations)

        # Build the transition matrix
        T = np.zeros((S, S, self.action_size))
        for action, outcomes in enumerate(self.action_prob):
            for outcome, prob in enumerate(outcomes):
                for prior_state in range(S):
                    post_state = neighbours[prior_state, outcome]
                    T[post_state, prior_state, action] += prob

        # Build the reward matrix
```

```
        R = self.default_reward * np.ones((S, S, self.action_size))
        for i, sr in enumerate(self.special_rewards):
            post_state = self.__loc_to_state__(locations, self.
                absorbing_locs[i])
            R[post_state, :, :] = sr

        return S, T, R, absorbing, locations, neighbours


    def __get_topology__(self):
        height = self.shape[0]
        width = self.shape[1]

        index = 1
        locs = []
        neighbour_locs = []

        for i in range(height):
            for j in range(width):
                # Get the location of each state
                loc = (i, j)

                # And append it to the valid state locations if it is a
                    valid state (ie. not absorbing)
                if self.__is_valid_location__(loc):
                    locs.append(loc)

                    # Get an array with the neighbours of each state, in
                        terms of locations
                    local_neighbours = [self.__get_neighbour__(loc,direction
                        ) for direction in self.action_names]
                    neighbour_locs.append(local_neighbours)

        # Translate neighbour lists from locations to states
        num_states = len(locs)
        state_neighbours = np.zeros((num_states, self.action_size), dtype=
            int)

        for state in range(num_states):
            for direction in range(self.action_size):
                # Turn neighbour location into a state number and insert it
                    into neighbour matrix
                state_neighbours[state, direction] = self.__loc_to_state__(
                    locs, neighbour_locs[state][direction])

        # Translate absorbing locations into absorbing state indices
        absorbing = list(map(lambda loc: self.__loc_to_state__(locs, loc),
            self.absorbing_locs))

        return locs, state_neighbours, absorbing


    def __loc_to_state__(self, locs, loc):
        # Takes list of locations and gives index corresponding to input
            location
        return locs.index(tuple(loc))


    def __is_valid_location__(self, loc):
        # It is a valid location if it is within grid shape and not an
```

```
        obstacle
    if loc[0] < 0 or loc[1] < 0 or loc[0] > self.shape[0] - 1 or loc[1]
        > self.shape[1] - 1:
            return False
    elif loc in self.obstacle_locs:
            return False

    return True


def __get_neighbour__(self, loc, direction):
    # Find valid neighbours that are in the grid and not an obstacle
    i = loc[0]
    j = loc[1]

    north = (i - 1, j)
    east = (i, j + 1)
    south = (i + 1, j)
    west = (i, j - 1)

    # If the neighbour is a valid location, accept it, otherwise, stay
        put
    if direction == "N" and self.__is_valid_location__(north):
            return north
    elif direction == "E" and self.__is_valid_location__(east):
            return east
    elif direction == "S" and self.__is_valid_location__(south):
            return south
    elif direction == "W" and self.__is_valid_location__(west):
            return west

    # Default is to return to the same location
    return loc


def __get_optimal_action__(self, Q):
    # Choose the optimal action that maxes Q for a given state
    optimal_actions = np.argwhere(Q == np.max(Q)).flatten()

    # If more than one action is optimal, randomly pick one
    if len(optimal_actions) > 1:
            optimal_action = random.choice(optimal_actions)
    else:
            optimal_action = optimal_actions[0]

    return optimal_action


def __epsilon_greedy__(self, policy, epsilon, Q):
    # Calculate the probability that we chose a random action to explore
    explore_prob = epsilon / policy.shape[1]

    # Set our policy to this probability across the policy
    epsilon_greedy_policy = np.full(policy.shape, explore_prob)

    # For each state in the policy
    for state in range(policy.shape[0]):
            # Choose the optimal action for the state based on Q
            optimal_action = self.__get_optimal_action__(Q[state])
```

```
            # Set the probability of our optimal action to be epsilon-greedy
            epsilon_greedy_policy[state, optimal_action] += 1 - epsilon

        return epsilon_greedy_policy


    # Generate an episode trace using exploring starts
    def __generate_episode__(self, policy):
        # Randomize the initial state from our non-absorbing states
        current_state = random.choice(self.non_absorbing)

        # Collect tuples of (state, action, reward) representing the trace
           for the episode
        episode = []

        # Until we reach an absorbing state
        while current_state not in self.absorbing:
            # Choose an action for the current state, given by our policy
            actions = policy[current_state]
            action = random.choice(len(actions), p=actions)

            # Randomize the outcome using our action-outcome matrix
            outcomes = self.action_prob[action]
            outcome = random.choice(len(outcomes), p=outcomes)

            # Find the successor state and determine our reward
            successor_state = self.neighbours[current_state, outcome]
            reward = self.R[successor_state, current_state, action]

            # Append the (state, action, reward) tuple to the episode
            episode.append((current_state, action, reward))

            # Repeat for the successor state
            current_state = successor_state

        # Append the terminal state to the episode without an action or
           reward
        episode.append((current_state, 0, 0))

        return episode


    ########################

    #### Methods ####

    def display(self):
        plt.figure()
        plt.subplot(1 ,3, 1)
        plt.imshow(self.walls)
        plt.subplot(1, 3, 2)
        plt.imshow(self.absorbers)
        plt.subplot(1, 3, 3)
        plt.imshow(self.rewarders)
        plt.show()


    def draw_deterministic_policy(self, policy):
        # Draw a deterministic policy
        # The policy needs to be a numpy array of 29 values between 0 and 3
```

```
            (0 -> N, 1 -> E, 2 -> S, 3 -> W)
        plt.figure()

        plt.imshow(self.walls + self.rewarders + self.absorbers)
        for state, action in enumerate(policy):
            if state not in self.absorbing:
                arrows = [r"$\uparrow$",r"$\rightarrow$", r"$\downarrow$", r
                    "$\leftarrow$"]
                action_arrow = arrows[np.argmax(action)]
                location = self.locs[state]
                plt.text(location[1], location[0], action_arrow, ha="center"
                    , va="center") # Place it on graph

        plt.show()


    def display_value_function(self, value):
        # Draw the given value function
        # The value needs to be a numpy array of 29 values
        plt.figure()
        plt.imshow(self.walls + self.rewarders + self.absorbers)

        for state, v in enumerate(value):
            if state not in self.absorbing:
                location = self.locs[state]
                plt.text(location[1], location[0], round(v, 3), ha="center",
                    va="center")

        plt.show()


    def value_iteration(self, gamma, threshold):
        # Get the transition and reward matrices
        T = self.T
        R = self.R

        epochs = 0
        delta = threshold # Set the value of delta as our stopping condition
        V = np.zeros(self.state_size) # Initialise value for each state to 0

        while delta >= threshold:
            epochs += 1
            delta = 0 # Reinitialise delta value

            # For each state
            for state in range(self.state_size):
                # If not an absorbing state
                if state not in self.absorbing:
                    # Store the previous value for that state
                    V_prev = V[state]

                    # Compute Q value
                    Q = np.zeros(self.action_size) # Initialise with value 0
                    for successor_state in range(self.state_size):
                        Q += T[successor_state, state, :] * (R[
                            successor_state, state, :] + gamma * V[
                            successor_state])

                    # Set the new value to the maximum of Q
                    V[state] = np.max(Q)
```

```
                # Compute the new delta
                delta = max(delta, np.abs(V_prev - V[state]))

    # When the loop is finished, fill in the optimal policy
    optimal_policy = np.zeros((self.state_size, self.action_size))

    # For each state
    for state in range(self.state_size):

        # Compute Q value
        Q = np.zeros(self.action_size)
        for successor_state in range(self.state_size):
            Q += T[successor_state, state, :] * (R[successor_state,
                state, :] + gamma * V[successor_state])

        # The action that maximises the Q value gets probability 1
        optimal_action = self.__get_optimal_action__(Q)
        optimal_policy[state, optimal_action] = 1

    return optimal_policy, V, epochs


def monte_carlo_iterative_optimisation(self, policy, episodes, epsilon,
    gamma, alpha):
    Q = np.zeros((self.state_size, self.action_size)) # Initialise value
        for each state-action to 0
    policy = self.__epsilon_greedy__(policy, epsilon, Q) # Use an
        epsilon-greedy policy based on Q
    optimal_value = np.zeros(self.state_size) # Initialise optimal value
        for each state to 0

    # Collect the cumulative rewards and optimal value function from
        each episode
    episode_rewards = np.zeros(episodes)
    episode_values = np.zeros((episodes, self.state_size))

    # For each episode
    for i in range(episodes):
        # Generate a random episode based on our policy and reverse it
            to facilitate computation
        # of total discounted return in each state along the trace
        trace = self.__generate_episode__(policy)
        trace.reverse()

        R = 0
        R_first = {} # Dictionary for holding the first-visit returns
            for each state in the trace
        for idx, transition in enumerate(trace[1:]):
            # Compute the total discounted return for each state and
                store it
            state, action, reward = transition
            R = gamma * R + reward
            R_first[(state, action)] = R

            # Calculate the cumulative rewards collected along the
                episode
            episode_rewards[i] += reward

        # For each transition in the trace
```

```
            for transition in trace[1:]:
                # Update our Q value of the state-action using the first-
                    visit return we sampled
                state, action = transition[0], transition[1]
                Q[state, action] += alpha * (R_first[(state, action)] - Q[
                    state, action])

            # Update our epsilon-greedy policy using the updated Q
            policy = self.__epsilon_greedy__(policy, epsilon, Q)

            # Calculate optimal V using the max Q for each state
            for state in range(self.state_size):
                optimal_value[state] = np.max(Q[state])

            episode_values[i] = optimal_value

        # After all episodes, fill in our optimal policy
        optimal_policy = np.zeros((self.state_size, self.action_size))

        for state in range(self.state_size):
            # The action that maximises the Q value gets probability 1
            optimal_action = self.__get_optimal_action__(Q[state])
            optimal_policy[state, optimal_action] = 1

        return optimal_policy, optimal_value, episode_rewards,
            episode_values

    def q_learning(self, episodes, epsilon, gamma, alpha):
        Q = np.zeros((self.state_size, self.action_size)) # Initialise value
            for each state-action to 0
        optimal_value = np.zeros(self.state_size) # Initialise optimal value
            for each state to 0

        # Collect the cumulative rewards and optimal value function from
            each episode
        episode_rewards = np.zeros(episodes)
        episode_values = np.zeros((episodes, self.state_size))

        # For each episode
        for i in range(episodes):
            # Randomize the initial state from our non-absorbing states
            current_state = random.choice(self.non_absorbing)

            # Until we reach an absorbing state
            while current_state not in self.absorbing:
                # Choose the optimal action for current state based on Q
                optimal_action = self.__get_optimal_action__(Q[current_state
                    ])

                # Calculate the probability that we chose a random action to
                     explore
                explore_prob = epsilon / self.action_size

                # Set an epsilon-greedy policy to choose the next action
                epsilon_greedy_policy = np.full(self.action_size,
                    explore_prob)
                epsilon_greedy_policy[optimal_action] += 1 - epsilon

                # Choose a random action using our epsilon-greedy policy
                action = random.choice(self.action_size, p=
```

```
                    epsilon_greedy_policy)

                # Randomize the outcome using our action-outcome matrix
                outcomes = self.action_prob[action]
                outcome = random.choice(len(outcomes), p=outcomes)

                # Find the successor state and determine our reward
                successor_state = self.neighbours[current_state, outcome]
                reward = self.R[successor_state, current_state, action]

                # Update our Q value using the reward and the successor
                    state-action's discounted Q value
                Q[current_state, action] += alpha * (reward + gamma * np.max
                    (Q[successor_state]) - Q[current_state, action])

                # Repeat for the successor state
                current_state = successor_state

                # Calculate the cumulative rewards collected along the
                    episode
                episode_rewards[i] += reward

                # Calculate optimal V using the max Q for each state
                for state in range(self.state_size):
                    optimal_value[state] = np.max(Q[state])

                episode_values[i] = optimal_value

        # After all episodes, fill in our optimal policy
        optimal_policy = np.zeros((self.state_size, self.action_size))

        for state in range(self.state_size):
            # The action that maximises the Q value gets probability 1
            optimal_action = self.__get_optimal_action__(Q[state])
            optimal_policy[state, optimal_action] = 1

        return optimal_policy, optimal_value, episode_rewards,
            episode_values


grid = GridWorld(p)
print(f"Creating the Grid World for p = {p}, represented as: \n")
grid.display()
print()

threshold = 0.00001

dp_optimal_policy, dp_optimal_value, epochs = grid.value_iteration(gamma,
    threshold)

print(f"Optimal value function computed using value iteration, after {epochs
    } epochs:\n")
grid.display_value_function(dp_optimal_value)
print()

print(f"Optimal policy computed using value iteration, after {epochs} epochs
    :\n")
grid.draw_deterministic_policy(dp_optimal_policy)
print()
```

```
# Investigate how the value of p affects our optimal value function and
    policy
p_values = [0.1, 0.25, 0.9]

for test_p in p_values:
    test_grid = GridWorld(test_p)

    print(f"For p = {test_p}, gamma = {gamma}:\n")
    dp_optimal_policy, dp_optimal_value, epochs = test_grid.value_iteration(
        gamma, threshold)

    print(f"Optimal value function computed using value iteration, after {
        epochs} epochs:\n")
    test_grid.display_value_function(dp_optimal_value)
    print()

    print(f"Optimal policy computed using value iteration, after {epochs}
        epochs:\n")
    test_grid.draw_deterministic_policy(dp_optimal_policy)
    print()

# Investigate how the value of gamma affects our optimal value function and
    policy
gamma_values = [0.25, 0.75]

for test_gamma in gamma_values:
    test_grid = GridWorld(p)

    print(f"For p = {p}, gamma = {test_gamma}:\n")
    dp_optimal_policy, dp_optimal_value, epochs = test_grid.value_iteration(
        test_gamma, threshold)

    print(f"Optimal value function computed using value iteration, after {
        epochs} epochs:\n")
    test_grid.display_value_function(dp_optimal_value)
    print()

    print(f"Optimal policy computed using value iteration, after {epochs}
        epochs:\n")
    test_grid.draw_deterministic_policy(dp_optimal_policy)
    print()


grid = GridWorld(p)
print(f"Creating the Grid World for p = {p}, represented as: \n")
grid.display()
print()

episodes = 1000
epsilon = 0.1
alpha = 0.05

# Initialise an initial unbiased policy
policy = np.full((grid.state_size, grid.action_size), 1 / grid.action_size)
print(f"Starting from a uniform policy with epsilon = {epsilon}, alpha = {
    alpha}:\n")

mc_optimal_policy, mc_optimal_value, mc_episode_rewards, mc_episode_values =
    grid.monte_carlo_iterative_optimisation(policy, episodes, epsilon, gamma
```

```
    , alpha)

print(f"Optimal value function computed using Monte Carlo iterative
    optimisation, over {episodes} episodes:\n")
grid.display_value_function(mc_optimal_value)
print()

print(f"Optimal policy computed using Monte Carlo iterative optimisation
    over, {episodes} episodes:\n")
test_grid.draw_deterministic_policy(mc_optimal_policy)
print()


repetitions = 1000
mc_learning_curves = np.zeros((repetitions, episodes))
mc_episode_values = np.zeros((repetitions, episodes, grid.state_size))

print(f"Learning curve obtained over {repetitions} repetitions of {episodes}
     episodes:\n")

# Repeat MC iterative optimisation to get a better estimate of our agent's
    learning curve
for i in range(repetitions):
    _, _, episode_rewards, episode_values = grid.
        monte_carlo_iterative_optimisation(policy, episodes, epsilon, gamma,
        alpha)
    mc_learning_curves[i] = episode_rewards
    mc_episode_values[i] = episode_values

# Determine the mean learning curve
mc_mean_learning_curve = np.mean(mc_learning_curves, axis=0)

# Determine the standard deviation of rewards
mc_std_learning_curve = np.std(mc_learning_curves, axis=0)

plt.figure(figsize=(15, 10))
plt.xlabel("No. of episodes")
plt.ylabel("Rewards")
plt.title("MC Learning Curve")

plt.plot(mc_mean_learning_curve, label="Mean")
plt.plot(mc_mean_learning_curve + mc_std_learning_curve, label="Mean + Std")
plt.plot(mc_mean_learning_curve - mc_std_learning_curve, label="Mean - Std")
plt.legend()

plt.savefig("figures/2c3.png")
plt.show()


# Investigate how the value of epsilon affects our agent's learning curve
epsilon_values = [0.01, 0.1]

fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.suptitle(r"Effect of $\epsilon$ on MC Learning Curve")

print("Investigating the effect of epsilon on the learning curve:\n")

for idx, test_epsilon in enumerate(epsilon_values):
    learning_curves = np.zeros((repetitions, episodes))
```

```
    for i in range(repetitions):
        _, _, episode_rewards, _ = grid.monte_carlo_iterative_optimisation(
            policy, episodes, test_epsilon, gamma, alpha)
        learning_curves[i] = episode_rewards

    mean_learning_curve = np.mean(learning_curves, axis=0)

    subplot = ax[idx]
    subplot.set(title=fr"$\epsilon$ = {test_epsilon}", xlabel="No. of
        Episodes", ylabel="Rewards")
    subplot.plot(mean_learning_curve)

plt.savefig("figures/2c4_epsilon.png")
plt.show()
print()

# Investigate how the value of alpha affects our agent's learning curve
alpha_values = [0.01, 0.1]

fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.suptitle(r"Effect of $\alpha$ on MC Learning Curve")

print("Investigating the effect of alpha on the learning curve:\n")

for idx, test_alpha in enumerate(alpha_values):
    learning_curves = np.zeros((repetitions, episodes))

    for i in range(repetitions):
        _, _, episode_rewards, _ = grid.monte_carlo_iterative_optimisation(
            policy, episodes, epsilon, gamma, test_alpha)
        learning_curves[i] = episode_rewards

    mean_learning_curve = np.mean(learning_curves, axis=0)

    subplot = ax[idx]
    subplot.set(title=fr"$\alpha$ = {test_alpha}", xlabel="No. of Episodes",
        ylabel="Rewards")
    subplot.plot(mean_learning_curve)

plt.savefig("figures/2c4_alpha.png")
plt.show()
print()


grid = GridWorld(p)
print(f"Creating the Grid World for p = {p}, represented as: \n")
grid.display()
print()

episodes = 1000
epsilon = 0.1
alpha = 0.05

td_optimal_policy, td_optimal_value, td_episode_rewards, td_episode_values =
    grid.q_learning(episodes, epsilon, gamma, alpha)

print(f"Optimal value function computed using Q-Learning, over {episodes}
    episodes:\n")
grid.display_value_function(td_optimal_value)
print()
```

```
print(f"Optimal policy computed using Q-Learning, over {episodes} episodes:\
    n")
test_grid.draw_deterministic_policy(td_optimal_policy)
print()


repetitions = 1000
td_learning_curves = np.zeros((repetitions, episodes))
td_episode_values = np.zeros((repetitions, episodes, grid.state_size))

print(f"Learning curve obtained over {repetitions} repetitions of {episodes}
    episodes:\n")

# Repeat Q-Learning to get a better estimate of our agent's learning curve
for i in range(repetitions):
    _, _, episode_rewards, episode_values = grid.q_learning(episodes,
        epsilon, gamma, alpha)
    td_learning_curves[i] = episode_rewards
    td_episode_values[i] = episode_values

# Determine the mean learning curve
td_mean_learning_curve = np.mean(td_learning_curves, axis=0)

# Determine the standard deviation of rewards
td_std_learning_curve = np.std(td_learning_curves, axis=0)

plt.figure(figsize=(15, 10))
plt.xlabel("No. of episodes")
plt.ylabel("Rewards")
plt.title("TD Learning Curve")

plt.plot(td_mean_learning_curve, label="Mean")
plt.plot(td_mean_learning_curve + td_std_learning_curve, label="Mean + Std")
plt.plot(td_mean_learning_curve - td_std_learning_curve, label="Mean - Std")
plt.legend()

plt.savefig("figures/2d3.png")
plt.show()


# Investigate how the value of epsilon affects our agent's learning curve
epsilon_values = [0.01, 0.1]

fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.suptitle(r"Effect of $\epsilon$ on TD Learning Curve")

print("Investigating the effect of epsilon on the learning curve:\n")

for idx, test_epsilon in enumerate(epsilon_values):
    learning_curves = np.zeros((repetitions, episodes))

    for i in range(repetitions):
        _, _, episode_rewards, _ = grid.q_learning(episodes, test_epsilon,
            gamma, alpha)
        learning_curves[i] = episode_rewards

    mean_learning_curve = np.mean(learning_curves, axis=0)

    subplot = ax[idx]
```

```
    subplot.set(title=fr"$\epsilon$ = {test_epsilon}", xlabel="No. of
       Episodes", ylabel="Rewards")
    subplot.plot(mean_learning_curve)

plt.savefig("figures/2d4_epsilon.png")
plt.show()
print()

# Investigate how the value of alpha affects our agent's learning curve
alpha_values = [0.01, 0.1]

fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.suptitle(r"Effect of $\alpha$ on TD Learning Curve")

print("Investigating the effect of alpha on the learning curve:\n")

for idx, test_alpha in enumerate(alpha_values):
    learning_curves = np.zeros((repetitions, episodes))

    for i in range(repetitions):
        _, _, episode_rewards, _ = grid.q_learning(episodes, epsilon, gamma,
            test_alpha)
        learning_curves[i] = episode_rewards

    mean_learning_curve = np.mean(learning_curves, axis=0)

    subplot = ax[idx]
    subplot.set(title=fr"$\alpha$ = {test_alpha}", xlabel="No. of Episodes",
        ylabel="Rewards")
    subplot.plot(mean_learning_curve)

plt.savefig("figures/2d4_alpha.png")
plt.show()
print()


def rms(array):
    return np.sqrt(np.mean(array ** 2))

mc_optimal_values = np.mean(mc_episode_values, axis=0)
td_optimal_values = np.mean(td_episode_values, axis=0)

# Compute the RMS error between the optimal value function and the MC / TD
   estimates
mc_mean_error = np.zeros(episodes)
td_mean_error = np.zeros(episodes)

for i in range(episodes):
    mc_mean_error[i] = rms(dp_optimal_value - mc_optimal_values[i])
    td_mean_error[i] = rms(dp_optimal_value - td_optimal_values[i])

plt.figure()
plt.title("Estimation Error vs. Episodes")
plt.xlabel("No. of Episodes")
plt.ylabel("Estimation Error")

plt.plot(mc_mean_error, label="MC Learning")
plt.plot(td_mean_error, label="TD Learning")
plt.legend()
```

```
plt.savefig("figures/2e1.png")
plt.show()


mc_m, mc_c = np.polyfit(mc_mean_error, mc_mean_learning_curve, 1)
td_m, td_c = np.polyfit(td_mean_error, td_mean_learning_curve, 1)

plt.figure()
plt.title("Estimation Error vs. Rewards")
plt.xlabel("Rewards")
plt.ylabel("Estimation Error")

plt.scatter(mc_mean_learning_curve, mc_mean_error)
plt.scatter(td_mean_learning_curve, td_mean_error)
plt.plot(mc_m * mc_mean_error + mc_c, mc_mean_error, "r", label="MC Learning
    ")
plt.plot(td_m * td_mean_error + td_c, td_mean_error, "g", label="TD Learning
    ")
plt.legend()

plt.savefig("figures/2e2.png")
plt.show()


plt.figure()
plt.title(r"Estimation Error vs. Episodes (Varying $\epsilon$)")
plt.xlabel("No. of Episodes")
plt.ylabel("Estimation Error")

print("Investigating the effect of epsilon on the estimation error over
    episodes:\n")

for test_epsilon in epsilon_values:
    mc_episode_values = np.zeros((repetitions, episodes, grid.state_size))
    td_episode_values = np.zeros((repetitions, episodes, grid.state_size))

    for i in range(repetitions):
        _, _, _, mc_episode_values[i] = grid.
            monte_carlo_iterative_optimisation(policy, episodes, test_epsilon
            , gamma, alpha)
        _, _, _, td_episode_values[i] = grid.q_learning(episodes,
            test_epsilon, gamma, alpha)

    mc_optimal_values = np.mean(mc_episode_values, axis=0)
    td_optimal_values = np.mean(td_episode_values, axis=0)

    # Compute the RMS error between the optimal value function and the MC /
        TD estimates
    mc_mean_error = np.zeros(episodes)
    td_mean_error = np.zeros(episodes)

    for i in range(episodes):
        mc_mean_error[i] = rms(dp_optimal_value - mc_optimal_values[i])
        td_mean_error[i] = rms(dp_optimal_value - td_optimal_values[i])

    plt.plot(mc_mean_error, label=fr"MC ($\epsilon$ = {test_epsilon})")
    plt.plot(td_mean_error, label=fr"TD ($\epsilon$ = {test_epsilon})")

plt.legend()
plt.savefig("figures/2e3_epsilon.png")
```

```
plt.show()
print()

plt.figure()
plt.title(r"Estimation Error vs. Episodes (Varying $\alpha$)")
plt.xlabel("No. of Episodes")
plt.ylabel("Estimation Error")

print("Investigating the effect of alpha on the estimation error over
    episodes:\n")

for test_alpha in alpha_values:
    mc_episode_values = np.zeros((repetitions, episodes, grid.state_size))
    td_episode_values = np.zeros((repetitions, episodes, grid.state_size))

    for i in range(repetitions):
        _, _, _, mc_episode_values[i] = grid.
            monte_carlo_iterative_optimisation(policy, episodes, epsilon,
            gamma, test_alpha)
        _, _, _, td_episode_values[i] = grid.q_learning(episodes, epsilon,
            gamma, test_alpha)

    mc_optimal_values = np.mean(mc_episode_values, axis=0)
    td_optimal_values = np.mean(td_episode_values, axis=0)

    # Compute the RMS error between the optimal value function and the MC /
        TD estimates
    mc_mean_error = np.zeros(episodes)
    td_mean_error = np.zeros(episodes)

    for i in range(episodes):
        mc_mean_error[i] = rms(dp_optimal_value - mc_optimal_values[i])
        td_mean_error[i] = rms(dp_optimal_value - td_optimal_values[i])

    plt.plot(mc_mean_error, label=fr"MC ($\alpha$ = {test_alpha})")
    plt.plot(td_mean_error, label=fr"TD ($\alpha$ = {test_alpha})")

plt.legend()
plt.savefig("figures/2e3_alpha.png")
plt.show()
print()
```