



Defining and Solving Reinforcement Learning Task

The goal of this assignment is to acquire experience in defining and solving a reinforcement learning (RL) environment, following Gym standards. The assignment consists of three parts. The first part focuses on defining an environment that is based on a Markov decision process (MDP). In the second part, we will apply a tabular method SARSA to solve an environment that was previously defined. In the third part, we apply the Double Q-learning method to solve a grid-world environment.

Part I: Define an RL Environment

In this part, we will define a grid-world reinforcement learning environment as an MDP. While building an RL environment, you need to define possible states, actions, rewards and other parameters.

Describe the environment that you defined. Provide a set of states, actions, rewards, main objective.

- In our environment, we have a lawnmower agent that has to reach the goal destination with maximum rewards.
- We have added two battery locations which count for positive rewards and two rock locations which damage the lawnmower as negative rewards.

To understand the locations in terms of grid indices, see the below code:

```
# start position of our agent
pos1 = [0, 0]

# goal position of our agent
pos2 = [3, 3]

# battery positions
b1 = [0, 3]
b2 = [2, 3]

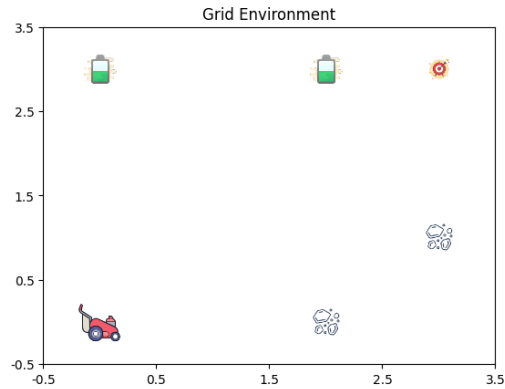
# ROck positions
r1 = [2, 0]
r2 = [3, 1]
```

- **Set of States:**

- We have 6 different states that a lawnmower agent can end up in:

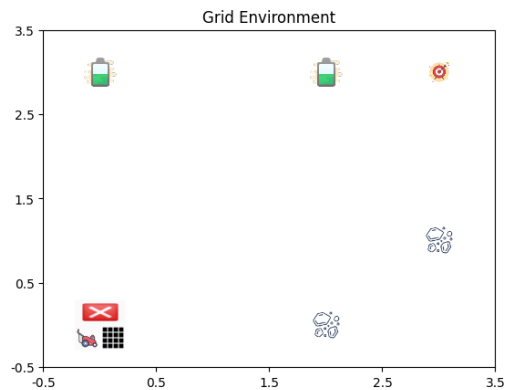
1) Agent Initial State: This is where the lawnmower starts his work to mow the garden grid.

- Initially he is at 0, 0 on the bottom left corner. The goal is at the top right corner of the grid, as shown below:



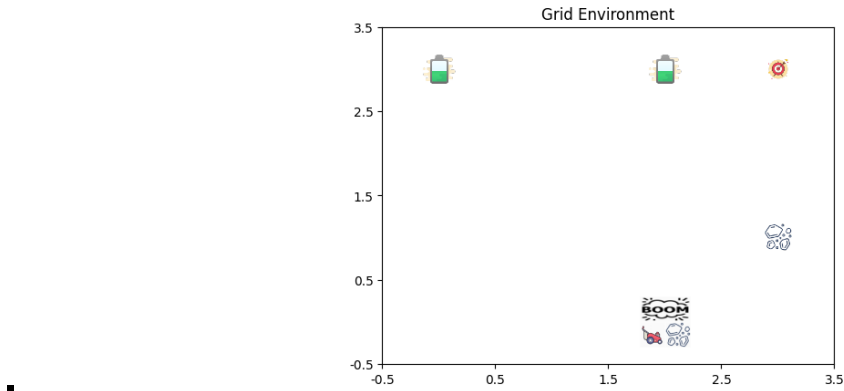
2) Agent going out of the Grid State: If the lawn mower tries to mow out of the grid he ends up in the state.

- Consider from the initial position, if the agent tries to go from 0,0 to -1, 0 or 0, -1 then the agent ends up in this going out of the grid state.



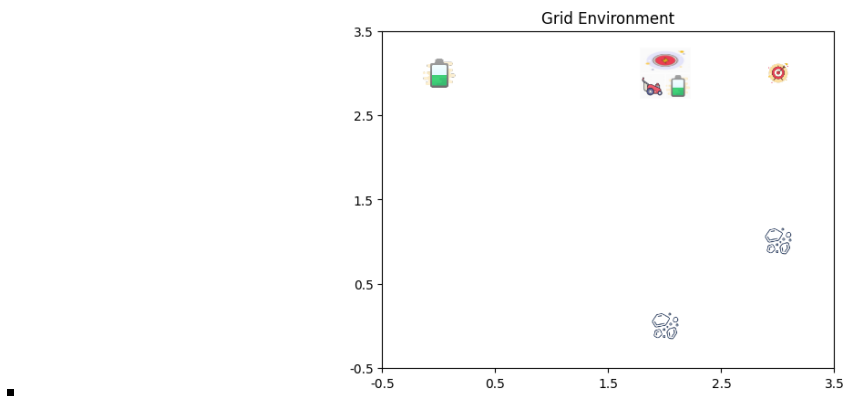
3) On Rocks State: If the agent lands on the rocks then he is in this state.

- Agent hits the rocks if he lands on the locations of the rocks are 2, 0 and 3, 1



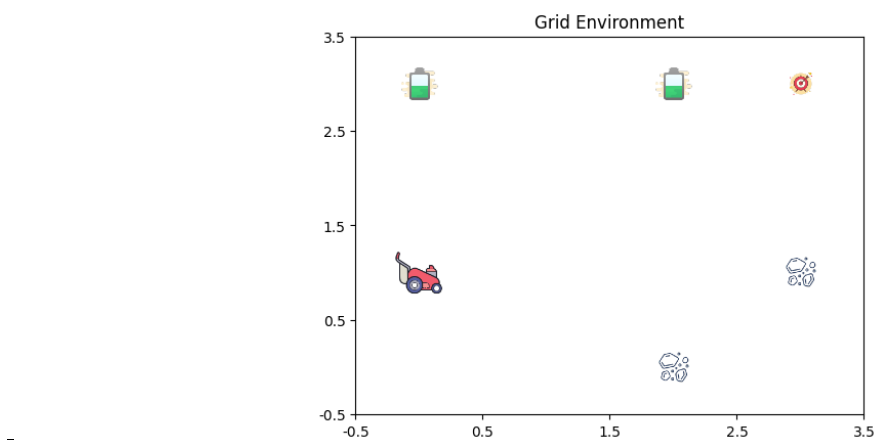
4) On Battery States: If the lawnmower/agent reaches the batteries he can charge himself and then actively reach the goal.

- The locations where the batteries are present is: 0, 3 and 2, 3



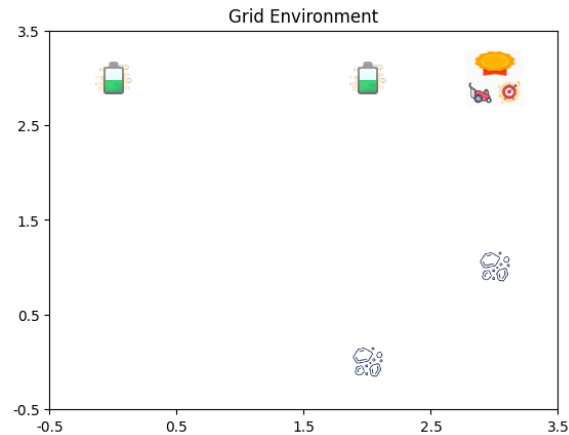
5) Intermediate State: Any other random location that the agent mows the garden grid is the intermediate state.

- Let he be present at 0, 1, then it is an intermediate state.



6) Goal State: The final goal state indicates the termination of the agents work as he has mowed the whole lawn.

- The goal state is 3,3 in the garden grid:



- **Set of Actions** defined in our case:

- We have 4 main actions which includes: Left, right, up and down.
- We also ensure to keep the agent within the grid by clipping over the boundaries and restoring his location from where he tried to move out of the grid.
- If the agent is in the same position as the previous step, choose a different action.
- The following code handles the direction movement and clipping:

```
if action == 0:
    self.myagent[0] += 1
elif action == 1:
    self.myagent[0] -= 1
elif action == 2:
    self.myagent[1] += 1
elif action == 3:
    self.myagent[1] -= 1

self.myagent = np.clip(self.myagent, 0, 3)

prev_state_positions = np.argwhere(self.prev_state == 0.2)
if len(prev_state_positions) > 0 and np.array_equal(self.myagent, prev_state_positions[0]):
    while action == self.prev_action:
        action = self.action_space.sample()
    self.flag_out = 1
```

- **Set of Rewards**

- **Goal Reached (Positive Reward):**
If the goal position is reached, the agent receives a reward of 10. This encourages the agent to reach the goal.
- **Rock Collision (Negative Rewards):**
If the agent collides with the first rock, the agent receives a penalty of -5. If the agent collides with the second rock, the agent receives a penalty of -6. These negative rewards discourage the agent from colliding with rocks.
- **Battery Charging (Positive Rewards):**
If the agent reaches the location of the first battery, the agent receives a reward of 5. If the agent reaches the location of the second battery, the agent receives a reward of 6. These positive rewards encourage the agent to reach and charge at battery locations.
- **Distance to Goal (Dynamic Rewards):**
If the current distance to the goal is less than the previous distance to the goal, the agent receives a reward of 1. This provides a positive reinforcement for moving closer to the goal.

If the current distance to the goal is greater than the previous distance to the goal, the agent receives a penalty of -1. This penalizes the agent for moving away from the goal.

If there is no change in distance, the agent receives a slight negative reward of -0.1. This also handles the case when he is trying to go out of the grid.

- The following snippet handles the rewards given to agent.

```
if np.array_equal(self.myagent, self.goal_loc):
    reward = 10
elif np.array_equal(self.myagent, self.rock_loc[0]):
    reward = -5
elif np.array_equal(self.myagent, self.rock_loc[1]):
    reward = -6
elif np.array_equal(self.myagent, self.battery_loc[0]):
    reward = 5
elif np.array_equal(self.myagent, self.battery_loc[1]):
    reward = 6
elif current_distance_to_goal < prev_distance_to_goal:
    reward = 1 # Positive reward for moving closer to goal
elif current_distance_to_goal > prev_distance_to_goal:
    reward = -1 # Negative reward for moving away to goal
else:
    reward = -0.1 # Slight negative reward for no change
```

- **Main Objective:**

- The primary objective is for the agent to learn a policy that maximizes the cumulative reward over time. The agent should navigate the grid, avoid obstacles (rocks), reach the goal efficiently, and utilize battery locations for charging when necessary. The dynamic rewards based on distance encourage the agent to find an optimal path to the goal.
- In our environment, the maximum reward that can be achieved in a single episode is determined by the structure of the reward system.

Reaching the Goal:

- The agent receives a reward of 10 for reaching the goal position ([3, 3]).

Collecting Batteries:

- The agent receives rewards for collecting batteries. Based on your reward structure, there are two batteries with rewards 5 and 6, respectively.

Avoiding Rocks:

- The agent should avoid rocks, as touching them results in negative rewards (-5 and -6).

Moving Towards the Goal:

- Each step towards the goal could potentially earn a reward of 1. However, the exact reward depends on whether each step objectively decreases the distance to the goal.

Based on this, the maximum reward in a single episode would be achieved by collecting both batteries and reaching the goal, while also making every move directly towards the goal. The maximum reward calculation would be something like this:

- Collecting both batteries: $5 + 6 = 11$
- Reaching the goal: 10
- Maximum steps moving towards the goal: This depends on the starting position and the path taken. Let's assume, for simplicity, that each step towards the goal earns a reward and the agent makes direct moves to the goal.
The number of steps to reach the goal in the most direct path from the starting position [0, 0] to [3, 3] is 6 steps (3 up and 3 right, or 3 right and 3 up). Therefore, the reward from steps towards the goal can be up to $6 * 1 = 6$.

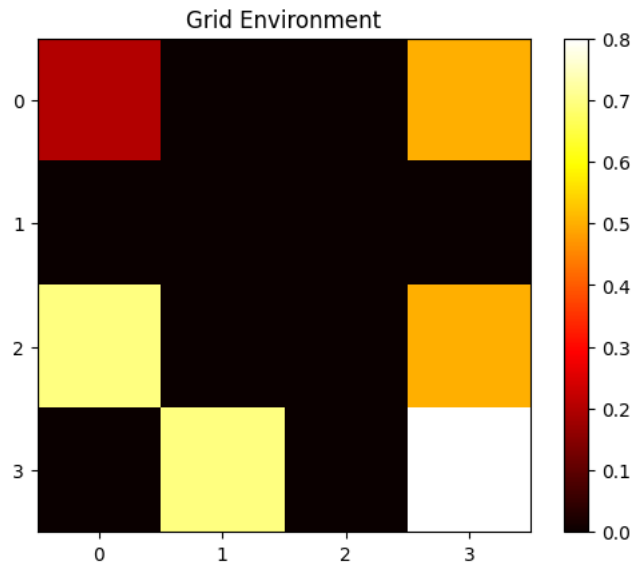
So, the theoretical maximum reward, assuming optimal play and direct pathing, could be:

▪ `11 (batteries) + 10 (goal) + 6 (steps towards the goal) = 27`

This is an ideal scenario and assumes the agent always takes the most direct path to the goal while collecting both batteries and not touching any rocks.

Provide visualization of your environment.

- Initially we defined our grid as shown below:
 - Heat map of grid as shown below:



Safety in AI: Write a brief review (~ 5 sentences) explaining how you ensure the safety of your environment. E.g. how do you ensure that the agent choose only actions that are allowed, that agent is navigating within defined state-space, etc

There are several measures implemented to ensure the safety of the environment and make sure the agent behaviour is under controlled.

- Firstly agent actions are restricted to a 4 moves that are up, down, left and right so it cant make any other actions like jump up-left corner or anything. Here's the example code snippet used

```
if action == 0:
    self.myagent[0] += 1
elif action == 1:
    self.myagent[0] -= 1
elif action == 2:
    self.myagent[1] += 1
elif action == 3:
    self.myagent[1] -= 1
self.myagent = np.clip(self.myagent, 0, 3)
```

- Second, we also implemented state-space constraints to ensure that our agent moves inside the grid and if the agent moves out or an attempt is made it sets a flag to indicate that agent is attempting to move out of the grid and negative penalty occurs of -0.1 because if he attempts to move out, he is put back to the same place and negative reward of -0.1 is applied.

```
truncated = True if np.any((self.myagent < 0) | (self.myagent > 3)) else False
if truncated:
```

```
self.flag_out = 1
```

- There's also a penalty for colliding with rocks, if it collides with rocks 1 a `reward = -5` or rock 2 a `reward = -6` and this is handled in the `calculate_reward` method.

```
penalty = any(np.array_equal(self.myagent, pos) for pos in self.rock_loc)
if penalty:
    self.penalty_counter += 1
```

- We are trying to reward him with positive points when the agent reaches batteries this recharges the agent.

```
elif np.array_equal(self.myagent, self.battery_loc[0]):
    reward = 5
elif np.array_equal(self.myagent, self.battery_loc[1]):
    reward = 6
```

- We are giving him positive rewards if he is taking steps towards the goal by verifying the distance from current position to goal position.

```
# Calculating distance to goal before and after the step
prev_distance_to_goal = np.linalg.norm(self.goal_loc - prev_myagentition)
current_distance_to_goal = np.linalg.norm(self.goal_loc - self.myagent)

elif current_distance_to_goal < prev_distance_to_goal:
    reward = 1 # Positive reward for moving closer to goal
elif current_distance_to_goal > prev_distance_to_goal:
    reward = -1 # Negative reward for moving away to goal
```

- To ensure that the task is completed by the agent and appreciated with 10 points as he reaches the goal.

```
terminated = True if np.array_equal(self.myagent, self.goal_loc) else self.timestep >= self.maximum_time
truncated = True if np.any((self.myagent < 0) | (self.myagent > 3)) else False
```

Part II: Implement SARSA

In this part, we implement SARSA (State-Action-Reward-State-Action) algorithm and apply it to solve the environment defined in Part 1. SARSA is an on-policy reinforcement learning algorithm. The agent updates its Q-values based on the current state, action, reward, and next state, action pair. It uses an exploration-exploitation strategy to balance between exploring new actions and exploiting the knowledge gained so far.

STEPS:

[Apply SARSA algorithm to solve the environment that was defined in Part I:](#)

- Lets understand step by step of SARSA implementation:

Formal definition: SARSA (State-Action-Reward-State-Action) is a temporal difference reinforcement learning algorithm that focuses on learning a policy for sequential decision-making in an environment.

- Initializing Parameters:

- Q-Table: we take a Q-table to store the expected cumulative rewards for each state-action pair.
- Exploration Rate (Epsilon): We set the initial exploration rate to encourage exploration in the early stages of learning.
- Discount Factor (Gamma): We choose a discount factor to balance immediate and future rewards.
- Learning Rate (Alpha): We set the learning rate to control the weight given to new information during updates.
- Epsilon Decay Rate (Decay Rate): We also took a decay rate to reduce exploration over time.
- Total Episodes: Determining the total number of episodes for training the agent.
- Initializing Tracking Variables:
 - Initialized lists (rewards_epi, epsilon_values, steps_per_episode, penalties_per_episode) to track rewards, exploration rate, steps, and penalties over episodes, which are required for our evaluation process.
- Main Training Loop over Episodes:
 - Loop over a predefined number of episodes.
 - Reset the environment to the initial state.
 - Convert the current observation to an index for accessing the Q-table.
- Exploration-Exploitation Tradeoff:
 - We choose an action based on the exploration-exploitation tradeoff with probability epsilon, explore by choosing a random action.
 - Otherwise, exploited the current policy by choosing the action with the highest Q-value for the current state.
- Execute Actions and Update Q-Table:
 - Executed the chosen action in the environment and observe the next state, the reward, and whether the episode terminated or was truncated.
 - Converted the next state to an index for accessing the Q-table.
 - Choose the next action based on the same exploration-exploitation tradeoff.
 - Update the Q-value for the current state-action pair using the SARSA update rule:


```
qt[state_index, action] = qt[state_index, action] + alpha * (reward + gamma * qt[next_strt_idx, next_action] - qt[state_index, action])
```
- Track Episode Statistics and check Episode Termination:
 - Update total rewards, total steps, and penalties for the current episode.
 - Check if the episode is terminated or truncated. If so, break out of the episode loop.
 - For Every 100 episodes, print and log the Q-table, average penalties, and average steps over the last 100 episodes.
- Epsilon Decay:
 - Decay the exploration rate (epsilon) to gradually reduce exploration over episodes.
- Final State:
 - store the final state of the environment after training.

```
env = MyLawn()

epsilon = 1.0 # Initial exploration rate
epsilon_min = 0.01 # Minimum exploration rate
gamma = 0.95 # Discount factor
alpha = 0.15 # Learning rate
decay_rate = 0.995 # Epsilon decay rate per episode
total_episodes = 1000
max_timestamp = 10

qt = np.zeros((env.obs_space.n, env.action_space.n))
```



```

rewards_epl = []
epsilon_values = []
steps_per_episode = []
penalties_per_episode = []

final_state = None
for episode in range(total_episodes):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_rewards = 0
    total_steps = 0
    action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[state_index])

    while True:
        next_state, reward, terminated, truncated, _ = env.step(action)
        total_steps += 1
        next_strt_idx = env.obs_space_to_index(next_state)
        next_action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[next_strt_idx])
        qt[state_index, action] = qt[state_index, action] + alpha * (reward + gamma * qt[next_strt_idx, next_action] - qt[state_index, action])
        state_index, action = next_strt_idx, next_action
        total_rewards += reward

        if terminated or truncated:
            break

    penalties_per_episode.append(env.get_penalty_count())

    # Q-table for every 100 episodes
    if (episode + 1) % 100 == 0:
        print(f"Episode: {episode + 1}")
        print("Q-table:")
        print(qt)

        # average penalties
        avg_penalty = np.mean(penalties_per_episode[-100:])
        print(f"Average Penalties in Last 100 Episodes: {avg_penalty}")

    epsilon = max(epsilon_min, epsilon * decay_rate)
    epsilon_values.append(epsilon)
    rewards_epl.append(total_rewards)
    steps_per_episode.append(total_steps)

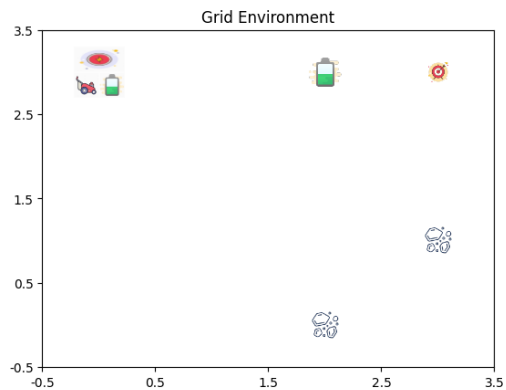
    if (episode + 1) % 100 == 0:
        average_steps = np.mean(steps_per_episode[-100:])
        print(f"Episode: {episode + 1}, Average Steps: {average_steps}")

    if episode == total_episodes - 1:
        final_state = env.state

env.state = final_state
env.render()

```

For our base model training we ended up in the following final state:



We had the following parameters defined for the Base Model training:

```
epsilon = 1.0 # Initial exploration rate
epsilon_min = 0.01 # Minimum exploration rate
gamma = 0.95 # Discount factor
alpha = 0.15 # Learning rate
decay_rate = 0.995 # Epsilon decay rate per episode
total_episodes = 1000
max_timestamp = 10
```

Base Model Results:

Provide the evaluation results:

- Print the initial Q-table and the trained Q-table
- Plot the total reward per episode graph (x-axis: episode, y-axis: total reward per episode).
- Plot the epsilon decay graph (x-axis: episode, y-axis: epsilon value)
- Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode.

a. We used the below snippet for getting the initial q-table:

```
initial_qt = np.zeros((env.obs_space.n, env.action_space.n))
print("Initial Q-table:")
print(initial_qt)
```

Output:

```
Initial Q-table:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

- Rows: Each row corresponds to a unique state in the environment. In our case, there are 16 possible states, representing different configurations of the lawn mower on the 4x4 grid.
- Columns: Each column corresponds to a specific action that the agent can take. In our case, there are four possible actions (down, up, right, left).

And we get the trained Q-Table as shown below:

```
print("\nTrained Q-table:")
```

```
print(qt)
```

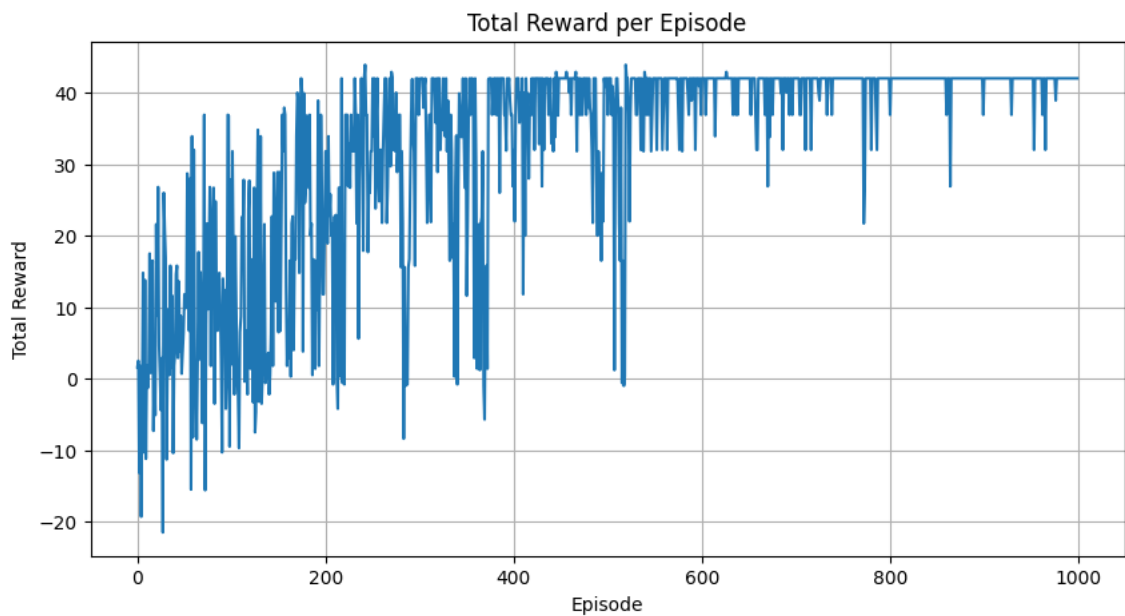
Output:

```
Trained Q-table:
[[ 5.91337632e+01  7.24750623e+01  8.09675919e+01  7.09371498e+01]
 [ 2.20130764e+01  4.28075140e+01  7.72520504e+01  5.03432100e+01]
 [ 3.28316495e+01  4.27783127e+01  8.06499444e+01  5.13757498e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.30973633e+01  5.67711941e+01  3.46939585e+00  3.34085173e+00]
 [ 9.44009470e-01  2.61848420e+00  2.98519490e+01  4.07660852e+00]
 [ 1.73511922e+01  1.44326895e+01  4.95925452e+01  2.62355274e+00]
 [ 7.19888261e+01  3.00369906e+01  2.93706064e+01  1.29366557e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.27797068e+00  1.55454431e-01  8.84338014e+00 -2.92968570e-01]
 [ 2.60250444e+00  1.41427035e+00  4.75767451e+01  6.00414188e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-9.51814050e-01 -1.80569118e+00 -2.29631598e+00 -9.73915099e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-5.39753466e-02 -1.66539461e-01  9.24575910e+00 -5.98452172e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. The total reward per episode is as follows:

```
# total reward per episode graph
plt.figure(figsize=(10, 5))
plt.plot(rewards_epi)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Total Reward per Episode')
plt.grid(True)
plt.show()
```

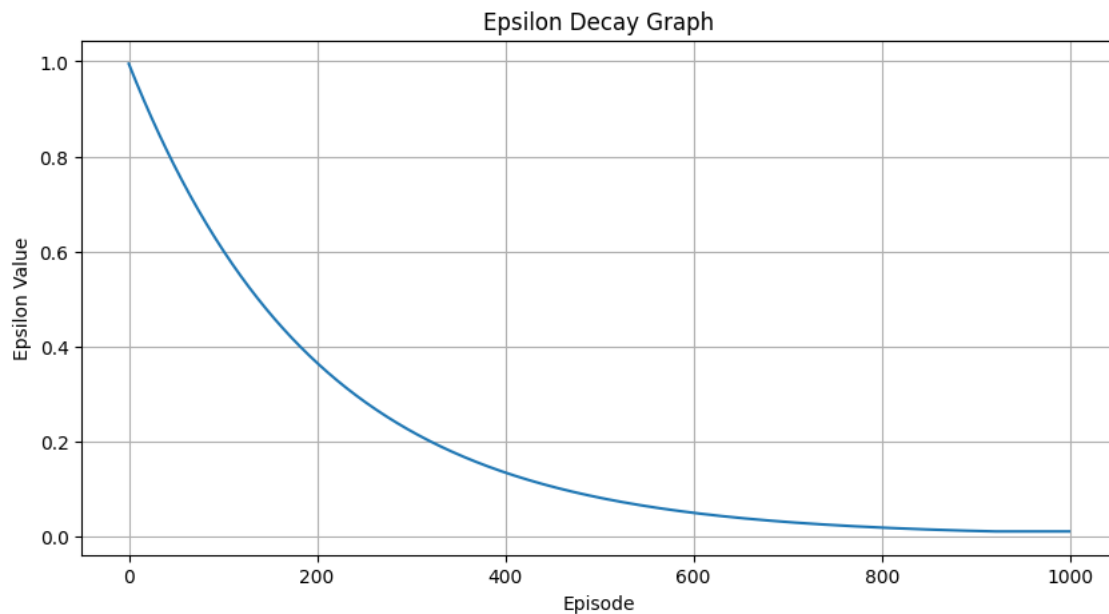
Output:



We ran over 1000 episodes for our base model, as we can see the total reward has fluctuated several times. But is mostly around 30 to 40 for episodes over 500.

c. The epsilon decay graph for our base model is as follows:

```
# epsilon decay graph
plt.figure(figsize=(10, 5))
plt.plot(epsilon_values)
plt.xlabel('Episode')
plt.ylabel('Epsilon Value')
plt.title('Epsilon Decay Graph')
plt.grid(True)
plt.show()
```



Epsilon is a parameter used in reinforcement learning systems to govern the trade-off between exploration and exploitation. Exploration is the act of attempting new things to learn about the environment, whereas exploitation is the process of maximizing our benefit by applying what we already know. By gradually decreasing epsilon over time, the lawnmower is able to learn about the environment and then exploit what it has learned to maximize its reward.

d. We executed the environment for 10 episodes, where the agent makes decisions based on the learned Q-table using a greedy policy.

```
greedy_rewards = []
for _ in range(10):
    total_reward = 0
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)

    while True:
        action = np.argmax(qt[state_index])
        next_state, reward, terminated, truncated, _ = env.step(action)
        next_strt_idx = env.obs_space_to_index(next_state)
        total_reward += reward
        state_index = next_strt_idx
        if terminated or truncated:
            print(f"Episode {episode + 1} Reward: {round(total_reward, 4)}")
            break
```

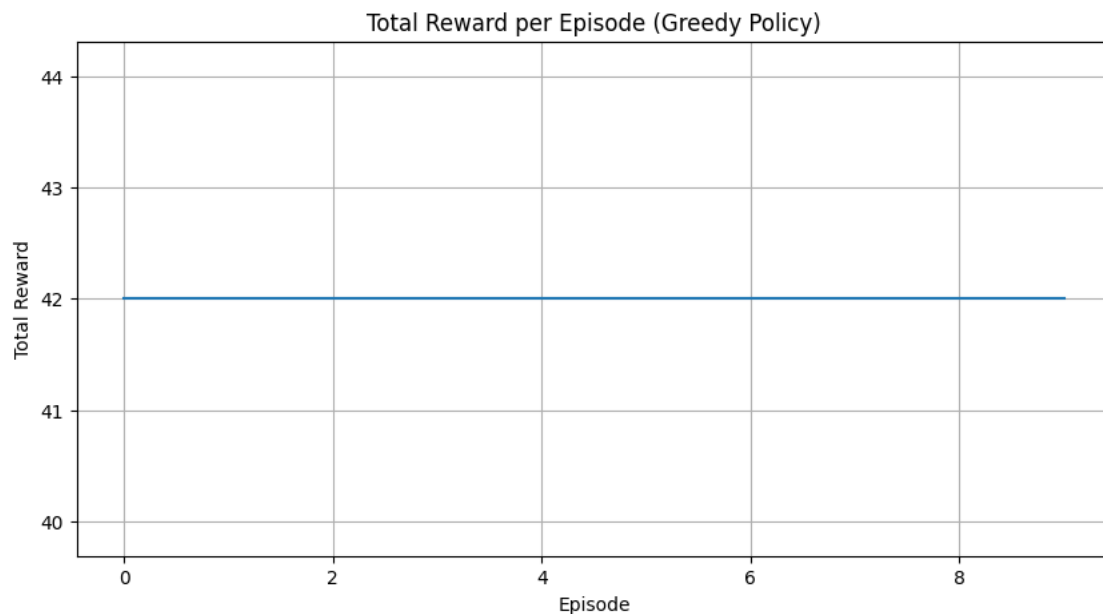
```

greedy_rewards.append(total_reward)

# total rewards for 10 episodes
plt.figure(figsize=(10, 5))
plt.plot(greedy_rewards)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Total Reward per Episode (Greedy Policy)')
plt.grid(True)
plt.show()

```

The total rewards obtained in each episode are then printed, and a plot is generated to visualize the performance of the greedy policy over the episodes. The greedy strategy selects the action with the highest Q-value in each state, aiming to exploit the learned policy for maximizing cumulative rewards.



We are getting the same output every episode because the agent always chooses greedy actions and the environment is deterministic. This means that the agent will always take the same actions in the same states, and the environment will always transition to the same next state and generate the same reward.

In the lawnmower grid environment, the environment is deterministic. This is because the next state and reward are only determined by the current state and action. Therefore, if the agent always chooses greedy actions, it will always get the same total reward per episode.

Hyperparameter tuning. Select at least two hyperparameters to tune to get better results for SARSA. You can explore hyperparameter tuning libraries, e.g. [Optuna](#) or make it Parameters to tune (select 2):

- Discount factor (γ)
- Epsilon decay rate
- Epsilon min/max values
- Number of episodes
- Maxtimesteps

Try at least 3 different values for each of the parameters that you choose.

We chose the Discount factor and Max timesteps hyperparameter tuning.

For the base model we had set: Gamma = 0.95 and Max_Time steps = 10

Now we tune and perform better using the following hyperparameters:

```
max_timestamp_values = [12, 15, 20]
gamma_values = [0.1, 0.5, 0.9]
```

I have defined a Training and evaluation loop as follows:

Training Loop:

```
performance_dict = {}

def training_loop(env, g, max_timestamp):
    alpha = 0.15 # Learning rate
    gamma = g # Discount factor
    epsilon = 1.0 # Initial exploration rate
    epsilon_min = 0.01 # Minimum exploration rate
    decay_rate = 0.995 # Epsilon decay rate per episode
    total_episodes = 1000
    max_timestamp = max_timestamp

    qt = np.zeros((env.obs_space.n, env.action_space.n))

    rewards_epi = []
    epsilon_values = []
    steps_per_episode = []
    penalties_per_episode = []

    final_state = None
    for episode in range(total_episodes):
        state, _ = env.reset()
        state_index = env.obs_space_to_index(state)
        total_rewards = 0
        total_steps = 0
        action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[state_index])

        while True:
            next_state, reward, terminated, truncated, _ = env.step(action)
            total_steps += 1
            next_strt_idx = env.obs_space_to_index(next_state)
            next_action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[next_strt_idx])
            qt[state_index, action] = qt[state_index, action] + alpha * (reward + gamma * qt[next_strt_idx, next_action] - qt[state_index, action])
            state_index, action = next_strt_idx, next_action
            total_rewards += reward

            if terminated or truncated:
                break

        penalties_per_episode.append(env.get_penalty_count())
        epsilon = max(epsilon_min, epsilon * decay_rate)
        epsilon_values.append(epsilon)
        rewards_epi.append(total_rewards)
        steps_per_episode.append(total_steps)

        if episode == total_episodes - 1:
            final_state = env.state

    final_reward = np.mean(rewards_epi[-100:])
    performance_dict[(alpha, gamma)] = final_reward
    return qt, rewards_epi, epsilon_values, final_state
```

Evaluation Loop:

```
def evaluate_loop(env, max_timestamp, gamma, qt, rewards_epi, epsilon_values, final_state):
    print(f"Max Timestamp, Gamma: {max_timestamp}, {gamma}")

    initial_qt = np.zeros((env.obs_space.n, env.action_space.n))
    print("Initial Q-table:")
    print(initial_qt)
    print("\nTrained Q-table:")
    print(qt)

    plt.figure(figsize=(10, 5))
    plt.plot(rewards_epi)
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')
    plt.title('Total Reward per Episode')
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(10, 5))
    plt.plot(epsilon_values)
    plt.xlabel('Episode')
    plt.ylabel('Epsilon Value')
    plt.title('Epsilon Decay Graph')
    plt.grid(True)
    plt.show()

    greedy_rewards = []
    for _ in range(10):
        total_reward = 0
        state, _ = env.reset()
        state_index = env.obs_space_to_index(state)

        while True:
            action = np.argmax(qt[state_index])
            next_state, reward, terminated, truncated, _ = env.step(action)
            next_strt_idx = env.obs_space_to_index(next_state)
            total_reward += reward
            state_index = next_strt_idx

            if terminated or truncated:
                print(f"Episode {episode + 1} Reward: {round(total_reward, 4)}")
                break

        greedy_rewards.append(total_reward)

    plt.figure(figsize=(10, 5))
    plt.plot(greedy_rewards)
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')
    plt.title('Total Reward per Episode (Greedy Policy)')
    plt.grid(True)
    plt.show()

    env.state = final_state
    env.render()
```

We have a nested loop where we are iterating over different values of **max_timestamp** and **gamma**. Inside the loop, we create a new instance of our environment (**MyLawn**) with specified values for **gamma**, and **max_timestamp**. Then, we are calling two functions, **training_loop** and **evaluate_loop**, passing the environment and other parameters to them.

Now using the below code:

```
max_timestamp_values = [12, 15, 20]
gamma_values = [0.1, 0.5, 0.9]
```

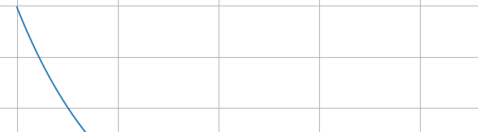
Provide the evaluation results (refer to Step 2) and your explanation for each result for each hyperparameter. In total, you should complete Step 2 seven times [Base model (step 1) + Hyperparameter #1 x 3 difference values & Hyperparameter #2 x 3 difference values]. Make your suggestion on the most efficient hyperparameters values for your problem setup

a. Q Tables:

[illegible]

```
[ [ 1.46970311  3.41440313  5.32299072  0.4405691 ]
[ 1.11133941  0.03625522  1.54657224 -0.57166931]
[ 1.13766417  0.41856681  5.4678709  -0.86114001]
[ 0.  0.  0.  0. ]
[-4.64129384 -0.72393866  1.10188127 -0.07464739]
[ 0.94367762 -0.92974816  1.16027634 -0.94522349]
[ 1.37074243 -0.65923108  1.642411  -0.92284503]
[ 6.48478471  5.39821022  0.46052814 -0.87588221]
[ 0.  0.  0.  0. ]
[-4.52491867 -0.73932166  1.47953695 -4.36248202]
[ 1.58159437 -0.68122192  6.4806719  -0.85483117]
[ 0.  0.  0.  0. ]
[-0.14717881 -2.28495063 -4.25782436 -0.15180451]
[ 0.  0.  0.  0. ]
[ 0.  -0.40017511  9.3687989  -2.27676613]
[ 0.  0.  0.  0. ]]
```

The plot shows the total reward per episode over 1000 episodes. The reward starts at approximately -10 at episode 0 and increases rapidly, reaching a plateau of about 40 by episode 200. After episode 200, the reward continues to fluctuate, with a slight upward trend towards 50, indicating that the agent is learning to perform better over time.



The graph, titled "Epsilon Decay Graph", illustrates the exponential decay of the epsilon value over 1000 episodes. The y-axis, labeled "Epsilon Value", ranges from 0.0 to 1.0. The x-axis, labeled "Episode", ranges from 0 to 1000. The curve starts at (0, 1.0) and decreases rapidly, reaching approximately 0.15 at episode 400, and then levels off, approaching 0.0 as the episode number reaches 1000.

| Episode | Epsilon Value |
|---------|---------------|
| 0 | 1.00 |
| 100 | 0.60 |
| 200 | 0.40 |
| 300 | 0.28 |
| 400 | 0.18 |
| 500 | 0.12 |
| 600 | 0.08 |
| 700 | 0.05 |
| 800 | 0.03 |
| 900 | 0.02 |
| 1000 | 0.01 |


```

Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52

```



For this hyperparameter combination we got 52 rewards as max timestamps are set to 12 and gamma value to 0.1. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 2:

Max Timestamp, Gamma: 12, 0.5

a. Q tables

Max Timestamp, Gamma: 12, 0.5

Initial Q-table:

```

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]

```

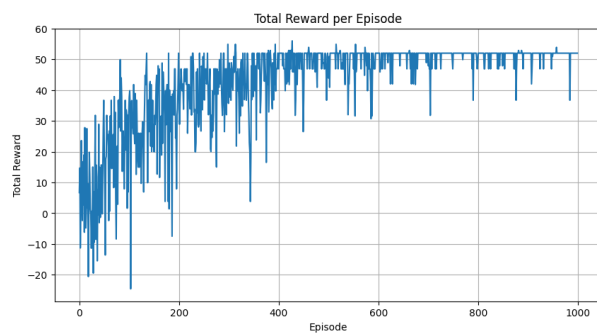
Trained Q-table:

```

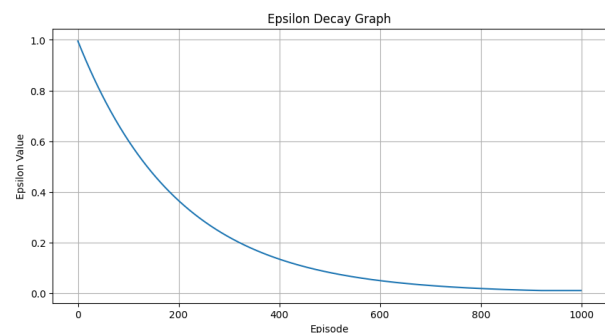
[[ 4.61725148  7.08450735  9.15103102  4.473967   ]
 [ 2.59008579  2.15723878  5.54128323  2.70413    ]
 [ 3.58942173  3.91348905  9.03929872  1.18799774 ]
 [ 0.          0.          0.          0.          ]
 [-2.91568935  1.89719398  2.59369217  0.2537576   ]
 [ 2.01322896  0.77907859  3.63389535 -0.98695131 ]
 [ 4.65975337  1.59361708  5.92828861 -0.22514045 ]
 [10.22553723  7.89707175  3.72449392  1.2584203   ]
 [ 0.          0.          0.          0.          ]
 [-2.45139486 -0.51677694  3.86059183 -2.05907993 ]
 [ 3.50364645  0.38252523  9.66127928  0.5260365   ]
 [ 0.          0.          0.          0.          ]
 [-0.50411383 -2.10640842 -3.46003095 -0.62823561 ]
 [ 0.          0.          0.          0.          ]
 [ 1.75206148  0.27565787 12.03210505 -2.69902898 ]
 [ 0.          0.          0.          0.          ]

```

b. total reward per episode



c. epsilon decay graph

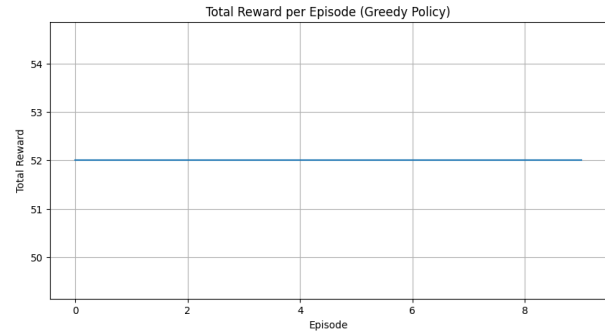


d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```

Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52

```



For this hyperparameter combination we got 52 again rewards as max timestamps are set to 12 and gamma value to 0.5. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 3:

Max Timestamp, Gamma: 12, 0.9

a. Q tables

```

Max Timestamp, Gamma: 12, 0.9
Initial Q-table:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]

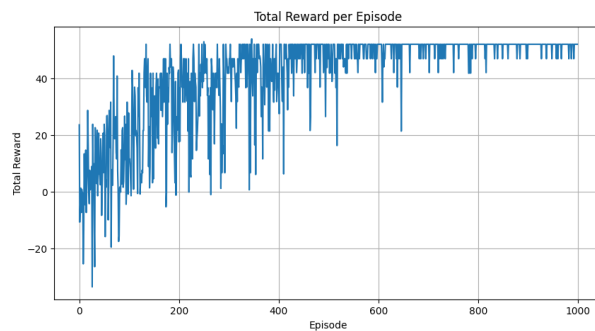
```

```

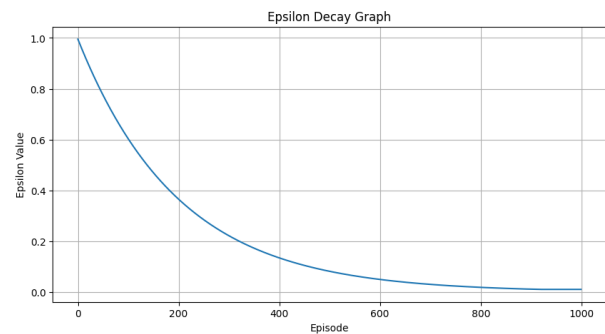
Trained Q-table:
[[36.37137631 40.62616894 42.84290847 37.63009399]
 [13.51990023 27.31724338 39.10379477 28.77798754]
 [21.42170126 28.92050144 42.38047933 25.8820195 ]
 [ 0.          0.          0.          0.          ]
 [ 2.78492105 29.19058035 10.9570313  3.29409681]
 [ 3.51261853  1.88664111 21.81663716  5.74846083]
 [ 4.93505003 17.10509433 30.57084824  6.00249408]
 [28.61756228 41.82683614 13.27949014 13.95625166]
 [ 0.          0.          0.          0.          ]
 [-3.19584955 -0.07401816  8.94806742  2.56293764]
 [ 0.61451853  4.23792876 18.12952044 -0.43065219]
 [ 0.          0.          0.          0.          ]
 [-0.51970308 -1.16001884 -3.45247113 -0.59830383]
 [ 0.          0.          0.          0.          ]
 [ 0.0910754  -0.07493813  3.92311504 -0.84924252]
 [ 0.          0.          0.          0.          ]

```

b. total reward per episode



c. epsilon decay graph

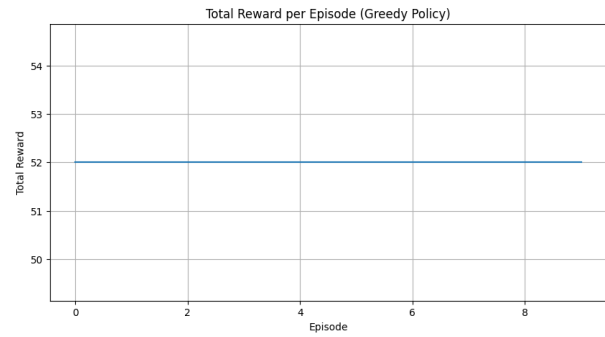


d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```

Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52
Episode 1000 Reward: 52

```



For this hyperparameter combination we got 52 rewards as max timestamps are set to 12 and gamma value to 0.9. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 4:

Max Timestamp, Gamma: 15, 0.1

a. Q tables

Max Timestamp, Gamma: 15, 0.1

Initial Q-table:

```

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]

```

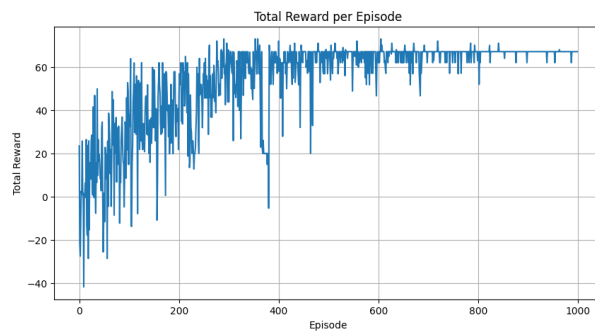
Trained Q-table:

```

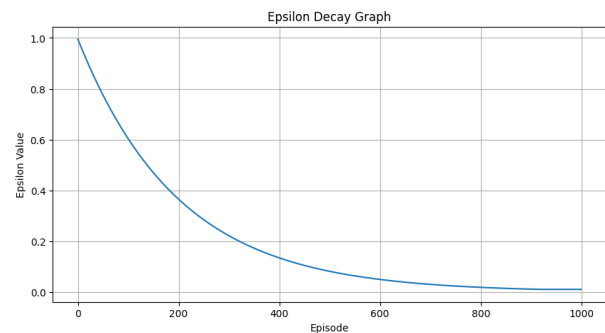
[[ 0.63387401  4.49217394  5.4194239   0.41588098]
 [ 1.09546548  0.03841747  1.5477699   -0.53358024]
 [ 1.13614283  0.38508015  5.47776281  -0.85850588]
 [ 0.          0.          0.          0.          ]
 [-4.66312364 -0.60161786  1.10611435 -0.08676839]
 [ 1.03731713 -0.86790714  1.14186436 -0.92941304]
 [ 1.45985031 -0.56884793  1.63914372 -0.9167606 ]
 [ 6.49988813  5.46777676  0.38329163 -0.8988462 ]
 [ 0.          0.          0.          0.          ]
 [-4.67021927 -0.82936068  1.36091674 -3.51612525]
 [ 1.47767054 -0.8515677   6.53779095 -0.8787274 ]
 [ 0.          0.          0.          0.          ]
 [-0.12683633 -2.74976258 -4.42604638 -0.20293501]
 [ 0.          0.          0.          0.          ]
 [-0.07407984 -0.60463816  9.50964965 -3.24711676]
 [ 0.          0.          0.          0.          ]]

```

b. total reward per episode



c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67
 Episode 1000 Reward: 67



For this hyperparameter combination we got 67 rewards as max timestamps are set to 15 and gamma value to 0.1. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 5:

Max Timestamp, Gamma: 15, 0.5

Max Timestamp, Gamma: 15, 0.5

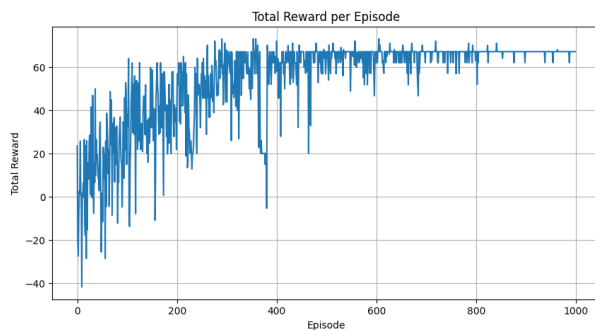
Initial Q-table:

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

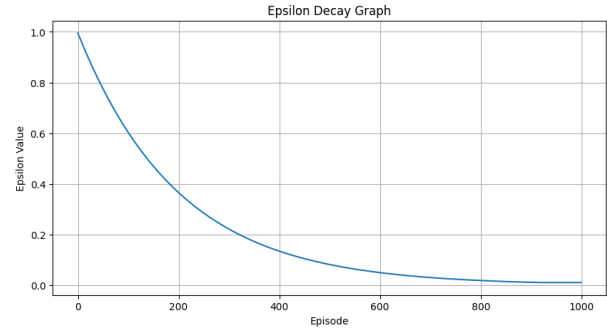
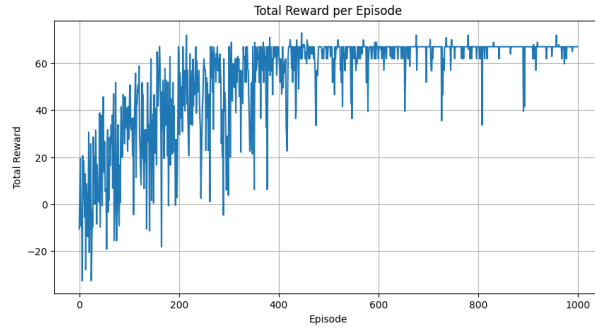
Trained Q-table:

```
[[ 6.10429441  7.04899434  9.45423241  4.55447072]
 [ 2.51726368  2.48898965  5.64105421  2.83352141]
 [ 3.83663186  3.92749272  9.29510376  1.30170597]
 [ 0.          0.          0.          0.          ]
 [-2.74620425  1.41327771  2.87533426  0.54192946]
 [ 1.98692623  0.64962254  3.92686809 -0.78439151]
 [ 6.15953464  2.13097004  4.54638811 -0.22250037]
 [10.08337894  8.73272162  3.60886446  1.08452078]
 [ 0.          0.          0.          0.          ]
 [-4.32065822 -0.70300552  4.89698408 -2.40639046]
 [ 4.20919439  0.93046564 10.4763109  -0.31382676]
 [ 0.          0.          0.          0.          ]
 [-0.57755291 -2.56681499 -3.82613055 -0.37305441]
 [ 0.          0.          0.          0.          ]
 [ 2.21839596  0.26612114 12.44067662 -2.37935666]
 [ 0.          0.          0.          0.          ]]
```

b. total reward per episode

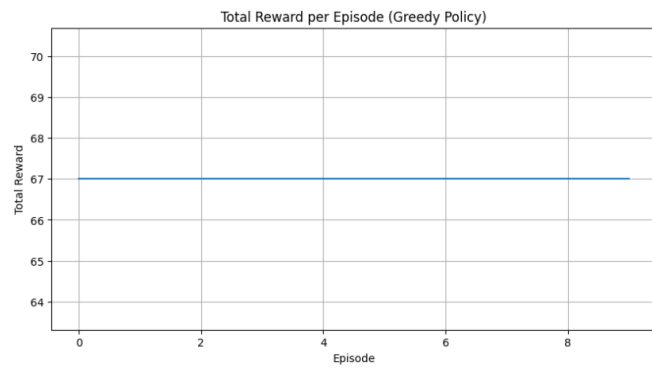


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
Episode 1000 Reward: 67
```



For this hyperparameter combination we got 67 rewards as max timestamps are set to 15 and gamma value to 0.9. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 7:

Max Timestamp, Gamma: 20, 0.1

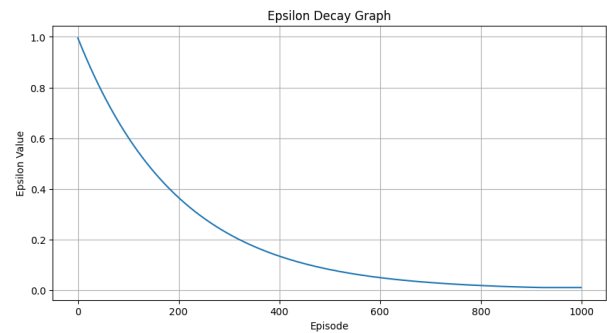
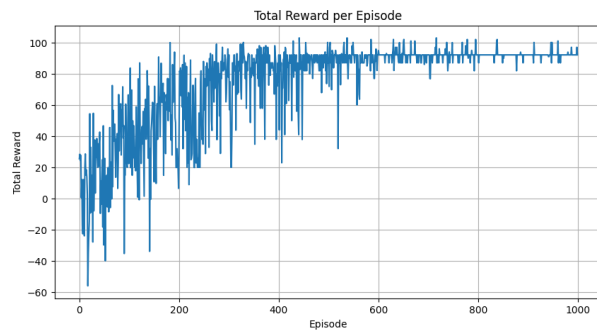
a. Q tables

```
Max Timestamp, Gamma: 20, 0.1
Initial Q-table:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
Trained Q-table:
[[ 0.6396296  3.97664547  5.55078612  0.46393006]
 [ 1.10818476  0.04236726  1.54893953 -0.53709752]
 [ 1.15369562  0.28471885  5.49625574 -0.86850648]
 [ 0. 0. 0. 0. ]
 [-4.69290749 -0.59429737  1.1063109 -0.15383971]
 [ 0.95183475 -0.89310294  1.15860392 -1.05697013]
 [ 1.37754243 -0.59560669  1.63701512 -0.94006684]
 [ 6.52064955  5.48490536  0.46375194 -0.88662793]
 [ 0. 0. 0. 0. ]
 [-5.58928188 -0.92285225  1.52190069 -4.36605764]
 [ 1.70133233 -0.8631555  6.53221304 -0.97364627]
 [ 0. 0. 0. 0. ]
 [-0.24811117 -4.35121215 -4.15815673 -0.23455997]
 [ 0. 0. 0. 0. ]
 [ 0.16666443 -0.69177423 10.31943178 -5.07697566]
 [ 0. 0. 0. 0. ]]
```

b. total reward per episode

c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
```



For this hyperparameter combination we got 92 rewards as max timestamps are set to 20 and gamma value to 0.1. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 8:

Max Timestamp, Gamma: 20, 0.5

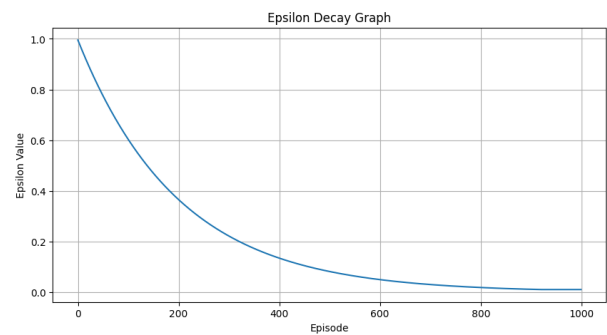
a. Q tables

```
Max Timestamp, Gamma: 20, 0.5
Initial Q-table:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
Trained Q-table:
[[ 4.8030952  7.48198186  9.67864823  4.37738969]
 [ 2.59379296  2.43739906  5.55230864  2.9817846 ]
 [ 3.50351528  4.12497918  9.44696107  1.11222408]
 [ 0.          0.          0.          0.          ]
 [-2.43700625  2.15066134  2.69506388 -0.01724466]
 [ 2.2431572  0.43983458  3.92492533 -0.17002617]
 [ 3.50106856  2.40809966  6.18926054  0.15830435]
 [10.63408843  8.94026861  4.58674334  1.22572239]
 [ 0.          0.          0.          0.          ]
 [-4.06557069 -0.50327173  5.57489487 -3.15083998]
 [ 3.74670625  0.7721339  10.42414932 -0.36155677]
 [ 0.          0.          0.          0.          ]
 [-0.82675286 -2.66075752 -4.07748227 -0.95917746]
 [ 0.          0.          0.          0.          ]
 [ 0.34433672  0.04122671 12.04536339 -2.75996367]
 [ 0.          0.          0.          0.          ]]
```

b. total reward per episode

c. epsilon decay graph

[illegible]

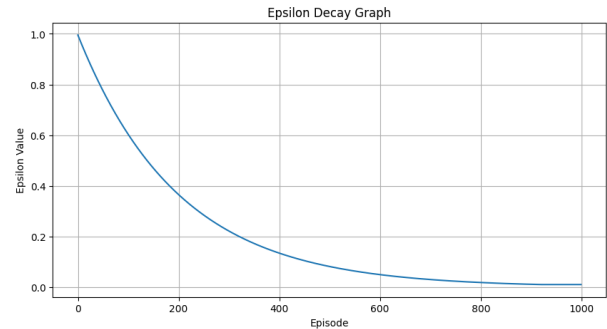
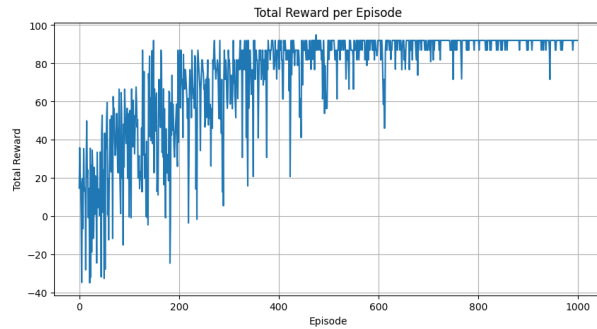
Hyperparameter Combination 9:

a. Q tables

[illegible]

```
Trained Q-table:
[[39.79210151 43.2525848 46.05511523 39.98431686]
 [19.83473043 32.58065465 41.77505309 33.15818029]
 [29.58515474 35.76642905 45.3066899 28.94020065]
 [ 0. 0. 0. 0. ]
 [ 8.99076725 30.50316918 5.93814289 13.06280766]
 [ 5.76677246 5.60641825 26.00683707 7.95489493]
 [15.02735801 20.55707935 36.85383526 5.81469757]
 [32.48022099 44.76087083 23.66505437 22.90242276]
 [ 0. 0. 0. 0. ]
 [ 0.69742182 1.35281669 14.6448358 4.49385335]
 [ 5.66028109 6.826243 27.65829931 2.84148434]
 [ 0. 0. 0. 0. ]
 [-1.09691353 -2.28453828 -1.32265503 -1.00740956]
 [ 0. 0. 0. 0. ]
 [ 0.30765731 3.71534286 18.73122875 5.21195961]
 [ 0. 0. 0. 0. ]]
```

c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
Episode 1000 Reward: 92
```



For this hyperparameter combination we got 92 rewards as max timestamps are set to 20 and gamma value to 0.9. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Suggestion on the best hyperparameters:

```
_, best_gamma, best_mt = max(performance_dict, key=performance_dict.get)
print(f"Best Time Stamp: {best_mt}, Best Gamma: {best_gamma}")
```

```
Best Time Stamp: 20, Best Gamma: 0.1
```

We got the most rewards and better results with time stamps 20 and gamma value = 0.1

Part III: Implement Double Q-learning

Apply Double Q-learning algorithm to solve the environment that was defined in Part I. You can modify your code from Part II

We implement Double Q-learning, where two Q-tables `qt1` and `qt2` are used to estimate the values of state-action pairs. We use the environment (MyLawn) over a specified number of episodes (total_episodes).

Initialization:

- env: An instance of the MyLawn environment.

- Hyperparameters:
 - epsilon: Initial exploration rate for epsilon-greedy policy.
 - epsilon_min: Minimum exploration rate.
 - gamma: Discount factor for future rewards.
 - alpha: Learning rate for updating Q-values.
 - decay_rate: Epsilon decay rate per episode.
 - total_episodes: The total number of episodes for training.
 - max_timestamp: Maximum number of timesteps allowed per episode.
- qt1: Q-table 1 initialization with zeros.
- qt2: Q-table 2 initialization with zeros.
- Lists for storing performance metrics:
 - rewards_epi: Total rewards obtained per episode.
 - epsilon_values: Values of epsilon over episodes.
 - steps_per_episode: Number of steps taken per episode.
 - penalties_per_episode: Number of penalties incurred per episode.

Training Loop:

- Iterates over episodes.
- Resets the environment and initializes episode-specific variables.
- Performs the main loop within an episode:
 - Chooses an action using epsilon-greedy policy based on the average of Q1 and Q2 values. Interact with the environment and receives the next state, reward, termination signal, and other information.
 - Update the chosen Q-table either `qt1` or `qt2` based on the observed reward and the maximum Q-value of the next state.
 - Continue the loop until termination, truncation, or reaching the maximum timestep.

Performance Monitoring:

- Record the number of penalties incurred during the episode.
- Every 100 episodes, prints and displays the Q-tables, average penalties, and average steps.
- Update epsilon based on the decay rate.
- Collect the final state of the environment after training.

The key idea of Double Q-learning is to mitigate overestimation bias by using two separate Q-tables for action selection and value estimation. The algorithm aims to learn more accurate Q-values and improve its policy over the training episodes.

```
env = MyLawn()
epsilon = 1.0
epsilon_min = 0.01
gamma = 0.95
alpha = 0.15
decay_rate = 0.995
total_episodes = 1000
max_timestamp = 10

qt1 = np.zeros((env.obs_space.n, env.action_space.n)) # Q-table 1 initialization
qt2 = np.zeros((env.obs_space.n, env.action_space.n)) # Q-table 2 initialization

rewards_epi = []
```

```

epsilon_values = []
steps_per_episode = []
penalties_per_episode = []

final_state = None

for episode in range(total_episodes):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_rewards = 0
    total_steps = 0

    while True:
        total_steps += 1
        action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax((qt1[state_index] + qt2[state_index]) / 2)
        next_state, reward, terminated, truncated, _ = env.step(action)
        next_strt_idx = env.obs_space_to_index(next_state)
        if np.random.uniform(0, 1) < 0.5:
            qt1[state_index, action] += alpha * (reward + gamma * qt2[next_strt_idx, np.argmax(qt1[next_strt_idx])]) - qt1[state_index, action])
        else:
            qt2[state_index, action] += alpha * (reward + gamma * qt1[next_strt_idx, np.argmax(qt2[next_strt_idx])]) - qt2[state_index, action])
        state_index = next_strt_idx
        total_rewards += reward
        if terminated or truncated or total_steps >= max_timestamp:
            break

    penalties_per_episode.append(env.get_penalty_count())
    if (episode + 1) % 100 == 0:
        print(f"Episode: {episode + 1}")
        print("Q-table 1:")
        print(qt1)
        print("Q-table 2:")
        print(qt2)
        avg_penalty = np.mean(penalties_per_episode[-100:])
        print(f"Average Penalties in Last 100 Episodes: {avg_penalty}")

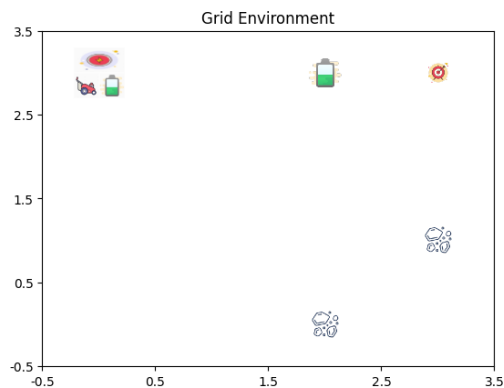
    epsilon = max(epsilon_min, epsilon * decay_rate)
    epsilon_values.append(epsilon)
    rewards_epi.append(total_rewards)
    steps_per_episode.append(total_steps)

    if (episode + 1) % 100 == 0:
        average_steps = np.mean(steps_per_episode[-100:])
        print(f"Episode: {episode + 1}, Average Steps: {average_steps}")

    if episode == total_episodes - 1:
        final_state = env.state

```

For our base model DOUBLE Q training we ended up in the following final state:



We had the following parameters defined for the Base Model training:

```

epsilon = 1.0
epsilon_min = 0.01
gamma = 0.95
alpha = 0.15
decay_rate = 0.995
total_episodes = 1000
max_timestamp = 10

```

Base Model Results:

Provide the evaluation results:

- Print the initial Q-table and the trained Q-table
- Plot the total reward per episode graph (x-axis: episode, y-axis: total reward per episode).
- Plot the epsilon decay graph (x-axis: episode, y-axis: epsilon value)
- Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode.

a. We used the below snippet for getting the q-tables:

```

print("Trained Q-table 1:")
print(qt1)
print("Trained Q-table 2:")
print(qt2)

```

Output:

```

Trained Q-table 1:
[[ 6.83981884e+01  8.09132391e+01  8.36755741e+01  7.63359833e+01]
 [ 3.85421682e+01  5.23848510e+01  8.07786252e+01  4.34273168e+01]
 [ 2.78350217e+01  6.58828656e+01  8.36648238e+01  5.20490177e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.50307932e+01  6.06983589e+01  1.10347637e+01  5.75602278e-01]
 [ 3.60057802e+00  5.83941186e+01  3.91609874e+00  3.11167087e+00]
 [ 1.34684638e+01  6.33693518e+00  4.98802604e+01  2.67137436e+00]
 [ 8.24392526e+01  8.79975389e+00  2.38981960e+01  1.69354427e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 3.33859471e+00 -2.21151040e-01  1.66946479e+01  2.84048081e+00]
 [ 2.95986567e+00  1.58132744e+00  3.35352283e+01 -1.90432541e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.50000000e-02  3.02565787e+00  0.00000000e+00 -1.31579062e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-4.01318437e-02 -2.81167245e-01  6.48924427e+00 -8.78838895e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

```

- Rows: Each row corresponds to a unique state in the environment. In our case, there are 16 possible states, representing different configurations of the lawn mower on the 4x4 grid.
- Columns: Each column corresponds to a specific action that the agent can take. In our case, there are four possible actions (down, up, right, left).

```

Trained Q-table 2:
[[ 7.49321166e+01  8.08305057e+01  8.27371432e+01  7.68169876e+01]
 [ 3.28862106e+01  5.52661461e+01  8.06232786e+01  5.73594119e+01]
 [ 3.76878799e+01  5.58731531e+01  8.42713195e+01  6.13537109e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 8.50621590e+00  5.82493544e+01  8.94841843e+00  2.46365356e+00]
 [ 4.18615404e+00  5.59757273e+01  1.51753013e+00  2.03239975e+00]
 [ 7.44551594e+00  3.91293574e+00  5.70541062e+01  1.01272710e+01]
 [ 8.09422102e+01  2.17238094e+01  1.43048770e+01  7.89188447e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 5.50081099e+00  6.07387055e+00  1.04857594e+01  2.88154181e+00]
 [ 1.41992013e+00  7.65647223e+00  5.49030165e+01  2.26151521e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-4.91120672e-02  1.58404938e+00  1.98215338e+00 -3.37500164e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.50000000e-02  0.00000000e+00  7.13987540e+00  4.35495458e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

```

b. The total reward per episode is as follows:

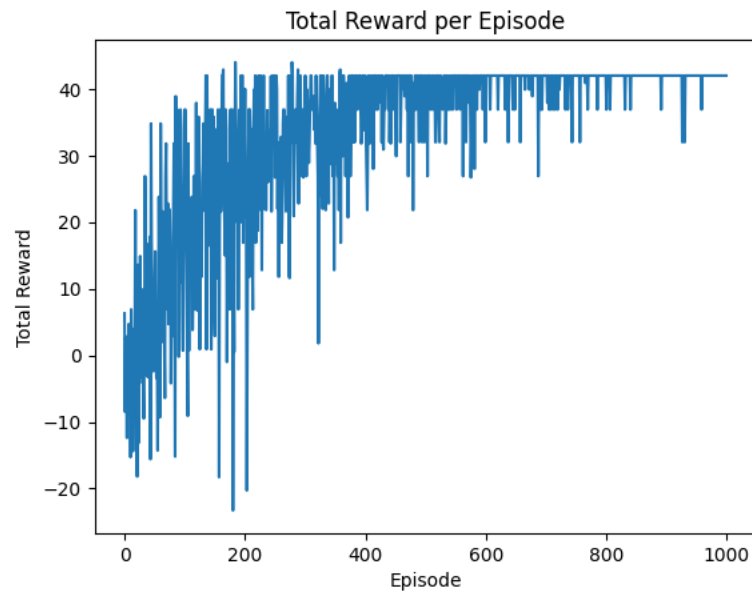
```

plt.plot(rewards_epi)
plt.title('Total Reward per Episode')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()

```

Output:

We ran over 1000 episodes for our base model, as we can see the total reward has fluctuated several times. But is mostly around 30 to 40 for episodes over 500.



c. The epsilon decay graph for our base model is as follows:

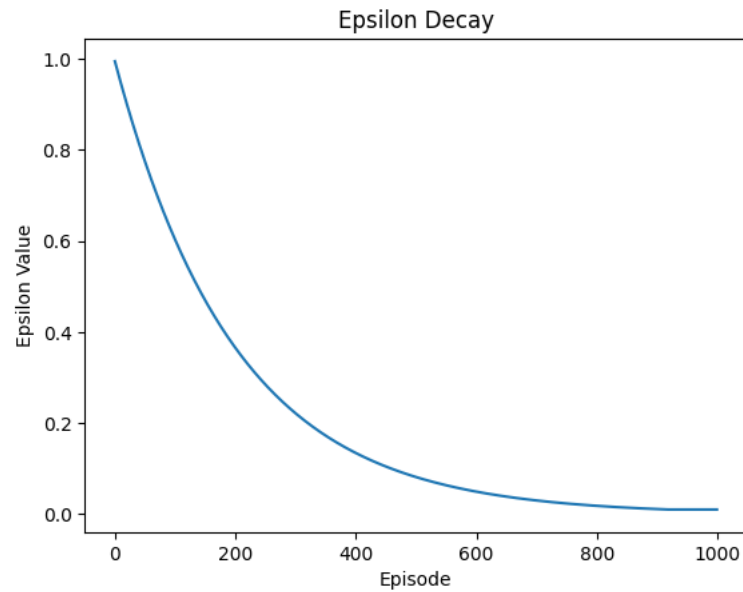
```

plt.plot(epsilon_values)
plt.title('Epsilon Decay')
plt.xlabel('Episode')

```

```
plt.ylabel('Epsilon Value')
plt.show()
```

By gradually decreasing epsilon over time, the lawnmower is able to learn about the environment and then exploit what it has learned to maximize its reward.



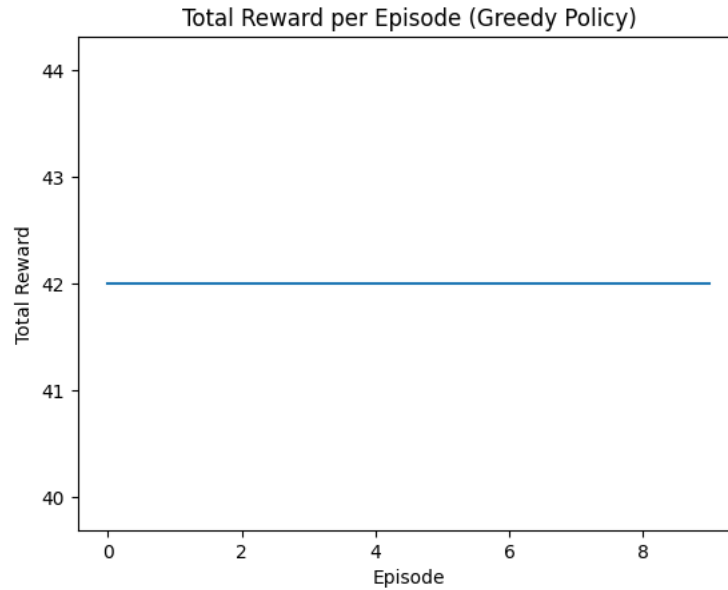
d. We executed the environment for 10 episodes, where the agent makes decisions based on the learned Q-table using a greedy policy.

```
greedy_rewards = []
for _ in range(10):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_greedy_rewards = 0
    while True:
        action = np.argmax((qt1[state_index] + qt2[state_index]) / 2)
        next_state, reward, terminated, truncated, _ = env.step(action)
        state_index = env.obs_space_to_index(next_state)
        total_greedy_rewards += reward

        if terminated or truncated:
            break

    greedy_rewards.append(total_greedy_rewards)

plt.plot(greedy_rewards)
plt.title('Total Reward per Episode (Greedy Policy)')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



We are getting the same output every episode because the agent always chooses greedy actions and the environment is deterministic. This means that the agent will always take the same actions in the same states, and the environment will always transition to the same next state and generate the same reward.

This is similar to SARSA.

Hyperparameter tuning. Select at least two hyperparameters to tune to get better results for Double Q-learning. You can explore hyperparameter tuning libraries, e.g. [Optuna](#) or make it Parameters to tune (select 2):

- Discount factor (γ)
- Epsilon decay rate
- Epsilon min/max values
- Number of episodes
- Maxtimesteps

Try at least 3 different values for each of the parameters that you choose.

We chose the Discount factor and Max timesteps hyperparameter tuning again this time to compare our SARSA AND Double Q Learning.

For the base model we had set: $\text{Gamma} = 0.95$ and $\text{Max_Time steps} = 10$

Now we tune and perform better using the following hyperparameters:

```
gamma_values = [0.9, 0.995, 0.99]
max_timestamp_values = [12, 15, 20]
```

I have defined a Training and evaluation loop as follows:

Training Loop:

```

performance_dict = {}

def training_loop(env, g , max_timestamp):
    gamma = g # Discount factor
    max_timestamp = max_timestamp
    epsilon = 1.0
    epsilon_min = 0.01
    alpha = 0.15
    decay_rate = 0.995
    total_episodes = 1000

    qt1 = np.zeros((env.obs_space.n, env.action_space.n))
    qt2 = np.zeros((env.obs_space.n, env.action_space.n))
    rewards_epi = []
    epsilon_values = []
    steps_per_episode = []
    penalties_per_episode = []
    final_state = None

    for episode in range(total_episodes):
        state, _ = env.reset()
        state_index = env.obs_space_to_index(state)
        total_rewards = 0
        total_steps = 0
        while True:
            total_steps += 1
            action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax((qt1[state_index] + qt2[state_index]) / 2)
            next_state, reward, terminated, truncated, _ = env.step(action)
            next_strt_idx = env.obs_space_to_index(next_state)
            if np.random.uniform(0, 1) < 0.5:
                qt1[state_index, action] += alpha * (reward + gamma * qt2[next_strt_idx, np.argmax(qt1[next_strt_idx])] - qt1[state_index, action])
            else:
                qt2[state_index, action] += alpha * (reward + gamma * qt1[next_strt_idx, np.argmax(qt2[next_strt_idx])] - qt2[state_index, action])

            state_index = next_strt_idx
            total_rewards += reward

            if terminated or truncated or total_steps >= max_timestamp:
                break

        penalties_per_episode.append(env.get_penalty_count())
        epsilon = max(epsilon_min, epsilon * decay_rate)
        epsilon_values.append(epsilon)
        rewards_epi.append(total_rewards)
        steps_per_episode.append(total_steps)

        if episode == total_episodes - 1:
            final_state = env.state

    avg_penalty = np.mean(penalties_per_episode[-100:])
    average_steps = np.mean(steps_per_episode[-100:])
    performance_dict[(gamma, max_timestamp)] = np.mean(rewards_epi[-100:])
    return rewards_epi, epsilon_values, avg_penalty, average_steps, qt1, qt2, final_state

```

Evaluation Loop:

```

def evaluate_loop(env, max_timestamp, gamma, qt1, qt2, rewards_epi, epsilon_values, final_state):
    print(f"Evaluation for gamma={gamma}, max_timestamp={max_timestamp}")

    print("Trained Q-table 1:")
    print(qt1)
    print("Trained Q-table 2:")
    print(qt2)

    plt.figure(figsize=(10, 5))
    plt.plot(rewards_epi)
    plt.title('Total Reward per Episode')
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')

```



```

plt.grid(True)
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(epsilon_values)
plt.title('Epsilon Decay')
plt.xlabel('Episode')
plt.ylabel('Epsilon Value')
plt.grid(True)
plt.show()

total_episodes = 10
greedy_rewards = []
for episode in range(total_episodes):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_rewards = 0

    while True:
        action = np.argmax((qt1[state_index] + qt2[state_index]) / 2) # Greedy action
        next_state, reward, terminated, truncated, _ = env.step(action)
        state_index = env.obs_space_to_index(next_state)
        total_rewards += reward

        if terminated or truncated:
            print(f"Episode {episode + 1} Reward: {round(total_rewards, 4)}")
            break

    greedy_rewards.append(total_rewards)
average_reward = np.mean(greedy_rewards)
print(f"Average Reward over {total_episodes} episodes: {average_reward}")

env.state = final_state
env.render()

```

We have a nested loop where we are iterating over different values of **max_timestamp** and **gamma**. Inside the loop, we create a new instance of our environment (**MyLawn**) with specified values for **gamma**, and **max_timestamp**. Then, we are calling two functions, **training_loop** and **evaluate_loop**, passing the environment and other parameters to them.

Now using the below code:

```

gamma_values = [0.9, 0.995, 0.99]
max_timestamp_values = [12, 15, 20]

for gamma in gamma_values:
    for mt in max_timestamp_values:
        env = MyLawn(gamma=gamma, alpha=0.15, max_timestamp=mt)
        print(f"Training for gamma={gamma}, max_timestamp={mt}")
        rewards_epi, epsilon_values, avg_penalty, average_steps, qt1, qt2, final_state = training_loop(env, gamma, mt)
        print(f"Average Reward: {np.mean(rewards_epi)}")
        print(f"Average Penalties: {avg_penalty}")
        print(f"Average Steps: {average_steps}")
        evaluate_loop(env, mt, gamma, qt1, qt2, rewards_epi, epsilon_values, final_state)

```

Provide the evaluation results (refer to Step 2) and your explanation for each result for each hyperparameter. In total, you should complete Step 2 seven times [Base model (step 1) + Hyperparameter #1 x 3 difference values & Hyperparameter #2 x 3 difference values]. Make your suggestion on the most efficient hyperparameters values for your problem setup

Hyperparameter Combination 1:

Training for gamma=0.9, max_timestamp=12

Average Reward: 43.295

Average Penalties: 0.0

Average Steps: 12.0

a. Q tables

Evaluation for gamma=0.9, max_timestamp=12

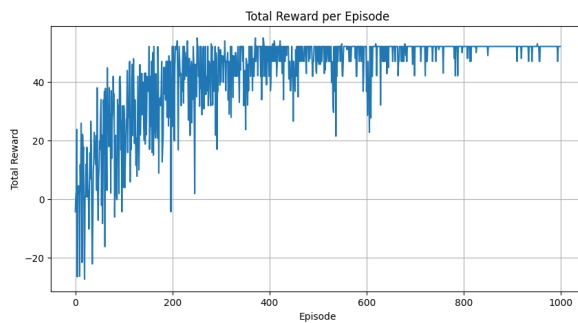
Trained Q-table 1:

```
[[ 3.22722066e+01  4.13886779e+01  4.34531864e+01  3.87109025e+01]
 [ 1.89714838e+01  2.76841088e+01  4.05271154e+01  2.79784841e+01]
 [ 2.70299107e+01  3.49160587e+01  4.37594888e+01  2.82786366e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 4.33196619e+00  1.12768202e+01  2.29585290e+01  9.94059563e-01]
 [ 6.15392807e+00  1.44586118e+01  3.00402625e+01  2.36076460e-01]
 [ 3.85891758e+01  1.61113615e+01  1.77984765e+01  2.67928587e-05]
 [ 4.41103011e+01  1.65849077e+01  1.20606576e+01  9.95049265e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-2.34380063e+00  2.62342810e+00  1.90896495e+01  1.14985914e+01]
 [ 4.69146035e+00  1.18770044e+01  4.18498693e+01  4.99622224e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-9.91684032e-02  -1.33288077e+00  -6.36434802e-01  -9.72159715e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  2.14748547e+00  1.51495278e+01  2.14783609e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

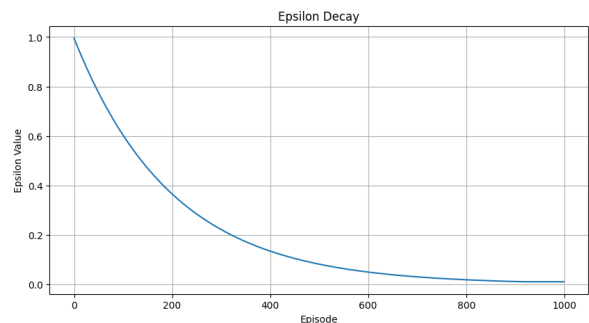
Trained Q-table 2:

```
[[ 3.89783749e+01  4.18402485e+01  4.38305941e+01  3.85532904e+01]
 [ 1.91935687e+01  2.65634903e+01  4.03256502e+01  3.30675464e+01]
 [ 2.94542432e+01  3.37347960e+01  4.41994053e+01  2.61984640e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 7.76466687e+00  1.27952449e+00  1.80525828e+01  1.89014463e+00]
 [ 5.93406133e+00  9.57610592e+00  3.22401251e+01  6.18538801e-01]
 [ 3.56207313e+01  1.47269809e+01  1.54269794e+01  8.31122561e+00]
 [ 4.33532294e+01  2.40032004e+01  8.78703199e+00  1.56026514e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 3.67778341e+00  3.95874944e+00  1.62203665e+01  5.34276920e-01]
 [ 8.19500998e+00  1.30308100e+01  4.37142342e+01  1.07063030e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.28667705e-01  -1.65910994e+00  -1.57282178e+00  -1.12242379e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.87794454e+00  3.10733510e+00  2.20235810e+01  -1.29188203e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. total reward per episode

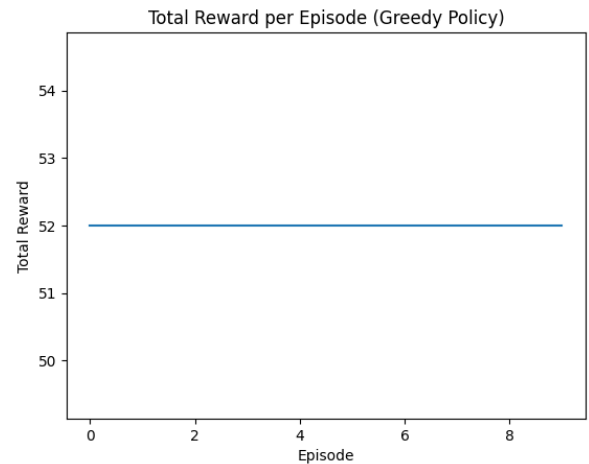


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 52
Episode 2 Reward: 52
Episode 3 Reward: 52
Episode 4 Reward: 52
Episode 5 Reward: 52
Episode 6 Reward: 52
Episode 7 Reward: 52
Episode 8 Reward: 52
Episode 9 Reward: 52
Episode 10 Reward: 52
Average Reward over 10 episodes: 52.0
```



For this hyperparameter combination we got 52 rewards as max timestamps are set to 12 and gamma value to 0.9. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 2:

Training for gamma=0.9, max_timestamp=15

Average Reward: 54.84539999999999

Average Penalties: 0.0

Average Steps: 15.0

a. Q tables

Evaluation for gamma=0.9, max_timestamp=15

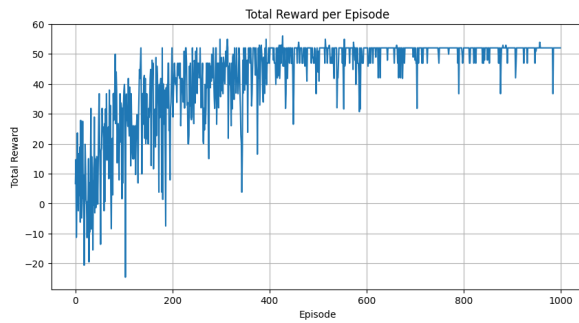
Trained Q-table 1:

```
[[ 3.84494869e+01  4.24833073e+01  4.46133789e+01  3.94878376e+01]
 [ 1.86080375e+01  2.89434871e+01  4.12475180e+01  3.12778551e+01]
 [ 2.54500129e+01  3.33930779e+01  4.48209580e+01  3.06504114e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 6.06091612e+00  3.32553271e+01  4.56574716e+00  4.30274985e+00]
 [ 5.95237430e+00  8.39008819e+00  2.53380954e+01  5.15425458e+00]
 [ 3.51181720e+01  1.01464013e+01  1.60754024e+01  2.60678265e+00]
 [ 3.02762720e+01  4.42982122e+01  2.27509732e+01  9.96271642e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 2.46002932e+00  3.80365109e-01  2.71424102e+01 -1.63272020e+00]
 [ 1.02676472e+01  8.87104662e+00  4.01989383e+01  5.37009266e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00 -3.61901876e-01  2.26593640e+00 -2.14529156e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  3.39745213e+00  2.97603462e+01 -8.87824687e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

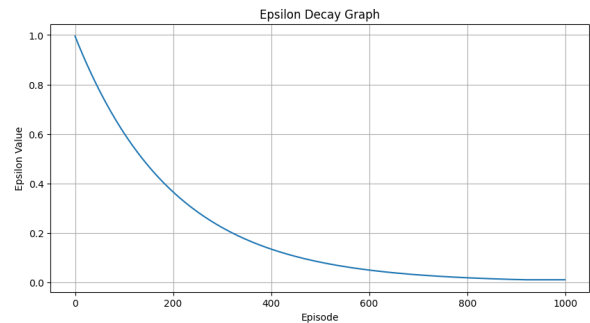
Trained Q-table 2:

```
[[35.90793313 42.3592176 44.97523612 39.53623059]
 [15.42287682 30.50377612 41.21109013 32.27486537]
 [27.39193137 33.48895392 44.68189902 30.41082117]
 [ 0. 0. 0. 0. ]
 [ 4.92920719 31.062453 1.30233339 5.28223108]
 [ 4.56554028 10.65679499 24.34169385 6.33269053]
 [33.77970597 18.75424563 16.99930084 3.10726476]
 [32.54347003 44.05061973 26.20750548 14.6445622 ]
 [ 0. 0. 0. 0. ]
 [ 2.64224825 0.16565014 10.67435963 6.15963216]
 [13.46976378 10.18073573 41.99724225 7.1362907 ]
 [ 0. 0. 0. 0. ]
 [-0.21310921 -0.34440902 1.01748452 -0.17739391]
 [ 0. 0. 0. 0. ]
 [ 1.23575349 6.27463266 17.28844616 6.62670853]
 [ 0. 0. 0. 0. ]]
```

b. total reward per episode



c. epsilon decay graph

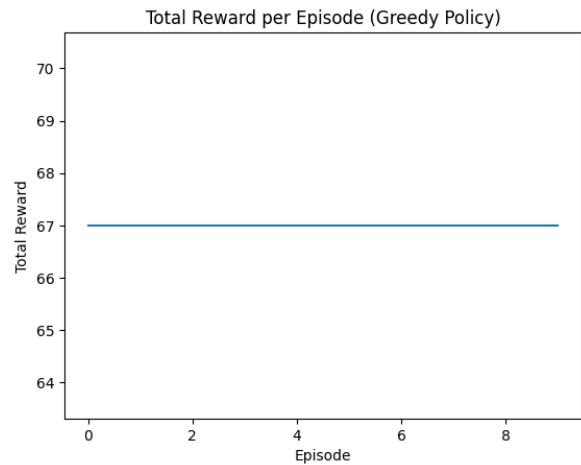


d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```

Episode 1 Reward: 67
Episode 2 Reward: 67
Episode 3 Reward: 67
Episode 4 Reward: 67
Episode 5 Reward: 67
Episode 6 Reward: 67
Episode 7 Reward: 67
Episode 8 Reward: 67
Episode 9 Reward: 67
Episode 10 Reward: 67
Average Reward over 10 episodes: 67.0

```



For this hyperparameter combination we got 67 rewards as max timestamps are set to 15 and gamma value to 0.9. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 3:

Training for gamma=0.9, max_timestamp=20

Average Reward: 75.1988

Average Penalties: 0.0

Average Steps: 20.0

a. Q tables

```

Evaluation for gamma=0.9, max_timestamp=20
Trained Q-table 1:
[[ 3.99625468e+01  4.45138381e+01  4.63271586e+01  4.10450600e+01]
 [ 2.32697316e+01  3.25580359e+01  4.25200053e+01  3.45462017e+01]
 [ 3.35245331e+01  3.32602749e+01  4.61999682e+01  2.92818020e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 7.23233404e+00  6.96224874e+00  3.02042309e+01  1.68942175e+00]
 [ 1.30241959e+01  1.22490854e+01  3.48973752e+01  7.58585493e+00]
 [ 1.75213322e+01  2.35320469e+01  4.11101987e+01  1.17872765e+01]
 [ 4.61225700e+01  3.65898587e+01  3.07905286e+01  2.61323858e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 7.34210764e+00  2.20625188e-01  2.97478635e+01  1.05769996e+01]
 [ 8.71677538e+00  2.08145056e+01  4.24859363e+01  1.09835901e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.50000000e-02  2.30174557e+00  2.42709630e+00  -2.62609576e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 3.73049278e+00  9.59580463e+00  1.42810087e+01  -2.13970721e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

```

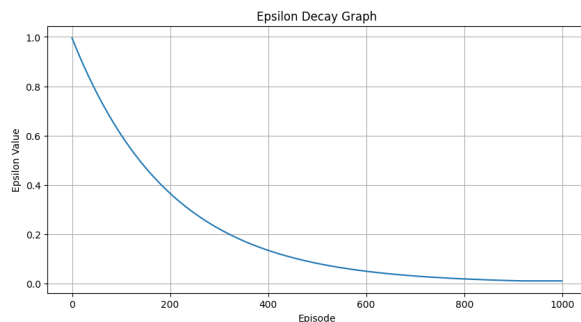
```

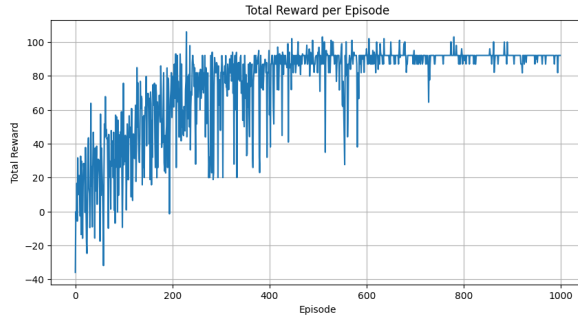
Trained Q-table 2:
[[ 3.93653014e+01  4.43812732e+01  4.64441654e+01  4.10917442e+01]
 [ 2.16183793e+01  2.93726828e+01  4.24589618e+01  3.18152475e+01]
 [ 3.18207651e+01  3.37023348e+01  4.60585376e+01  3.15641132e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.55308936e+01  1.08651314e+01  2.78402171e+01  6.32455935e+00]
 [ 5.00076209e+00  1.47851658e+01  3.56731987e+01  1.01844831e+00]
 [ 2.36102689e+01  2.16106099e+01  3.96270245e+01  1.36160233e+01]
 [ 4.68143156e+01  3.25464299e+01  3.57160219e+01  2.59340174e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.13052146e+01  6.15900696e-01  2.70866106e+01  8.71171014e+00]
 [ 8.47744840e+00  1.69223507e+01  4.33949813e+01  8.73035047e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-9.09124844e-01  8.64002888e+00  8.69268894e-01  -6.42232013e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-2.77500000e-02  8.99247263e+00  2.57546175e+01  4.42526297e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

```

b. total reward per episode

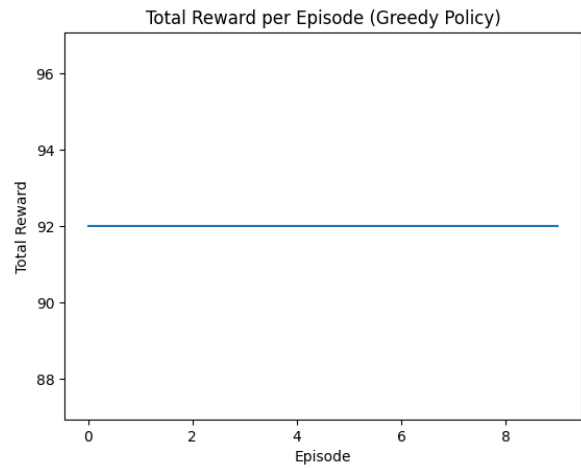
c. epsilon decay graph





d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 92
Episode 2 Reward: 92
Episode 3 Reward: 92
Episode 4 Reward: 92
Episode 5 Reward: 92
Episode 6 Reward: 92
Episode 7 Reward: 92
Episode 8 Reward: 92
Episode 9 Reward: 92
Episode 10 Reward: 92
Average Reward over 10 episodes: 92.0
```



For this hyperparameter combination we got 92 rewards as max timestamps are set to 20 and gamma value to 0.9. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 4:

Training for gamma=0.995, max_timestamp=12

Average Reward: 43.2045

Average Penalties: 0.0

Average Steps: 12.0

a. Q tables

Evaluation for gamma=0.995, max_timestamp=12

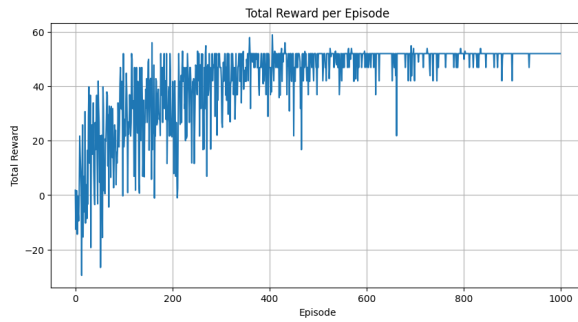
Trained Q-table 1:

```
[[ [ 3.53746379e+02  4.40919758e+02  5.80338532e+02  4.47898738e+02]
 [ 4.83760402e+01  1.33657438e+02  5.68886530e+02  2.35675395e+02]
 [ 1.02713408e+02  1.60508214e+02  5.74912493e+02  2.13600174e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 3.53811278e+00  2.69381415e+02  2.35389082e+00  4.83866044e+00]
 [ 4.03470344e+00  1.52849974e+02  1.39126918e+01  5.84967437e+00]
 [ 2.06554605e+01  1.74410503e+01  1.73762305e+02  1.24280360e+01]
 [ 4.53629630e+02  7.42471581e+01  3.20686770e+01  1.79041064e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 8.21701452e+00  1.04971833e+00  1.51541714e+01  9.31991859e+00]
 [ 1.30060133e+00  1.33746957e+01  8.31937246e+01  2.70912950e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-9.12165159e-02  3.93620155e-01  1.49627374e-01  -4.73931936e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 2.92019712e-01  0.00000000e+00  1.35505525e+01  -8.14904067e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

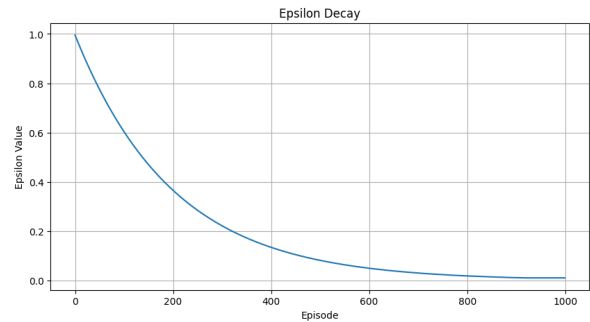
Trained Q-table 2:

```
[[ [ 3.35624260e+02  4.63941780e+02  5.80080962e+02  4.44877362e+02]
 [ 3.46855904e+01  1.59473794e+02  5.69466569e+02  2.95245740e+02]
 [ 8.63482192e+01  1.65762723e+02  5.77434838e+02  2.16780041e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 8.48187634e+01  2.04355532e+02  1.75856458e+00  1.25543107e+01]
 [ 4.88083240e+00  1.08277296e+02  1.49330136e+01  1.69380505e+00]
 [ 8.40992551e+00  4.50309158e+01  2.19749702e+02  8.41386985e-01]
 [ 4.15568131e+02  5.54299922e+01  7.62267883e+01  3.77613859e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.01318096e+01  1.15317477e+01  1.55659588e+00]
 [ 3.70563258e+00  1.76817063e+01  8.52613764e+01  3.80707079e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-6.38775974e-02  -8.75886227e-01  -2.40752438e-01  -4.08262500e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.83638729e+00  2.17690164e+01  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. total reward per episode

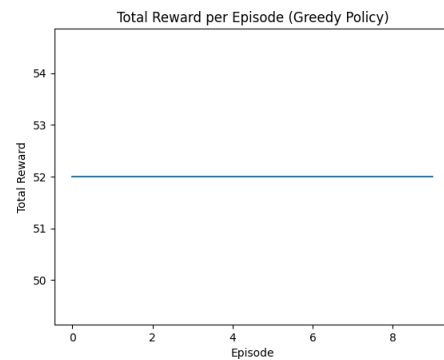


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 52
Episode 2 Reward: 52
Episode 3 Reward: 52
Episode 4 Reward: 52
Episode 5 Reward: 52
Episode 6 Reward: 52
Episode 7 Reward: 52
Episode 8 Reward: 52
Episode 9 Reward: 52
Episode 10 Reward: 52
Average Reward over 10 episodes: 52.0
```



For this hyperparameter combination we got 52 rewards as max timestamps are set to 12 and gamma value to 0.995. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 5:

Training for gamma=0.995, max_timestamp=15

Average Reward: 23.6874

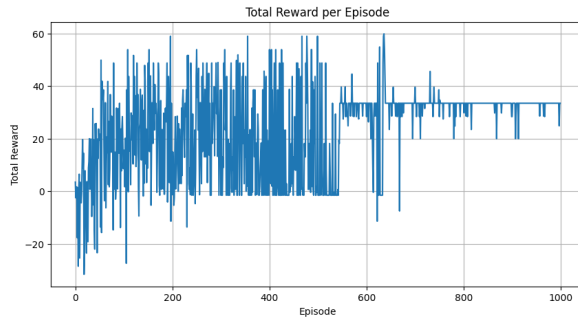
Average Penalties: 0.0

Average Steps: 14.75

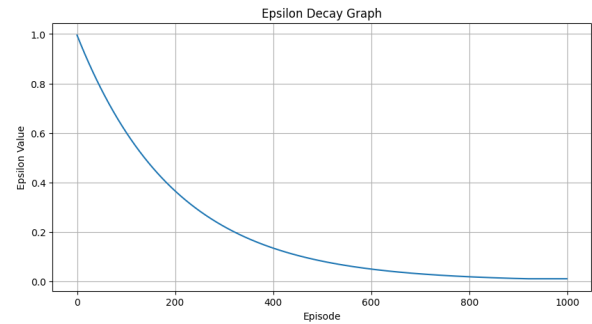
```
Evaluation for gamma=0.995, max_timestamp=15
Trained Q-table 1:
[[ 2.47707947e+02  1.79127156e+02  1.59600385e+02  1.34991412e+02]
 [ 1.47680958e+02  3.63057836e+01  4.70163462e+01  1.74358218e+01]
 [ 1.72021663e+01  1.50197836e+01  1.47886950e+02  1.70619402e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 5.36844804e+01  8.15324700e+01  2.36782023e+02  4.54874232e+01]
 [ 3.94581120e+01  7.44525088e+01  2.40229415e+02  5.43829519e+01]
 [ 5.30681177e+01  5.45148132e+01  2.47562891e+02  1.15678663e+02]
 [ 1.43610757e+02  2.45805092e+02  1.30854733e+02  1.28149654e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 2.56922337e+01  1.55903330e+01  6.25795618e+01  1.14021945e+01]
 [ 4.07638216e+00  1.30527305e+01  1.36054811e+02  1.27709188e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-4.72978963e-01  6.94474193e+01  -2.96143633e+00  -6.64852036e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 6.98978710e-01  8.03366730e-02  2.33538709e+01  5.83564333e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

```
Trained Q-table 2:
[[ 2.42965787e+02  1.72918416e+02  1.44572129e+02  1.52591825e+02]
 [ 1.51318306e+02  2.33308979e+01  3.55642986e+01  3.57586427e+01]
 [ 2.35359935e+01  4.34810912e+01  1.77228903e+02  1.20069717e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 4.52343465e+01  4.45853905e+01  2.38904718e+02  1.03912905e+02]
 [ 3.88757178e+01  7.46270330e+01  2.42550601e+02  8.46990548e+01]
 [ 4.81315313e+01  7.05767306e+01  2.41946968e+02  7.06587113e+01]
 [ 1.84116625e+02  2.50924051e+02  1.61603914e+02  1.18841432e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 9.26952710e+00  2.48068679e+00  8.07346364e+01  1.74871537e+00]
 [ 1.03421116e+01  2.41119496e+01  1.37154732e+02  6.82192005e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.76589075e+00  7.63159061e+01  3.65608062e+00  -1.39169724e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.72387500e-02  -2.40756082e-01  1.17908845e+01  4.82003061e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. total reward per episode

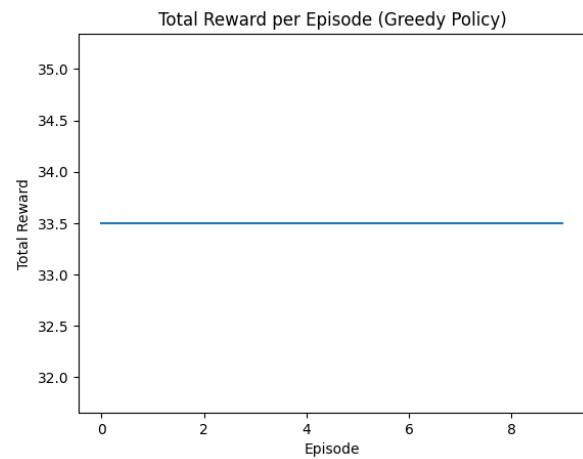


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 33.5
Episode 2 Reward: 33.5
Episode 3 Reward: 33.5
Episode 4 Reward: 33.5
Episode 5 Reward: 33.5
Episode 6 Reward: 33.5
Episode 7 Reward: 33.5
Episode 8 Reward: 33.5
Episode 9 Reward: 33.5
Episode 10 Reward: 33.5
Average Reward over 10 episodes: 33.5
```



For this hyperparameter combination we got 33.5 rewards as max timestamps are set to 15 and gamma value to 0.995. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 6:

Training for gamma=0.995, max_timestamp=20

Average Reward: 66.5946

Average Penalties: 0.0

Average Steps: 20.0

a. Q tables

Evaluation for gamma=0.995, max_timestamp=20

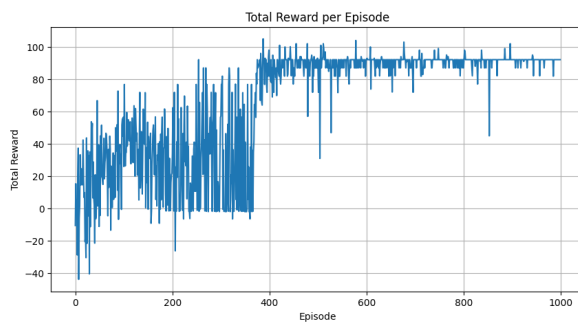
Trained Q-table 1:

```
[[ [ 4.90336486e+02  6.51358428e+02  7.39460194e+02  6.08951838e+02]
 [ 9.21502134e+01  2.59382522e+02  7.26313263e+02  1.17155051e+02]
 [ 7.81346394e+01  2.81871525e+02  7.32546584e+02  3.09594165e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.06780513e+02  2.72841310e+02  4.78005502e+01  8.99820117e+01]
 [ 4.22593726e+00  2.69431911e+02  4.62627289e+01  1.85341317e+01]
 [ 2.20319637e+01  2.07246932e+01  2.45746806e+02  1.67748880e+01]
 [ 6.24119390e+02  7.52469722e+01  1.40732324e+02  7.15408159e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 5.19864593e+00  8.86558686e+00  2.77370133e+01  7.90073652e+00]
 [ 9.39138158e+00  1.13873853e+01  2.32941165e+02  4.45311878e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ -6.01103814e-01  3.20597255e+01  3.96728969e-01  -6.46519360e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 4.02011356e+00  3.92458007e+00  2.53250981e+01  -4.16353675e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

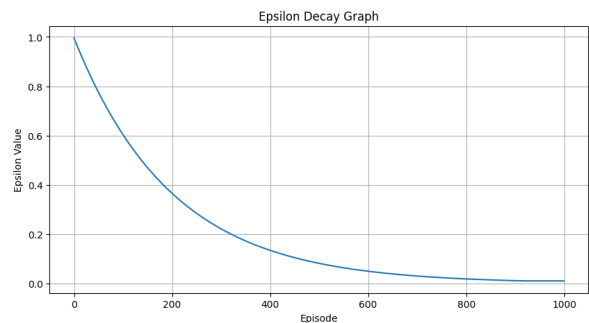
Trained Q-table 2:

```
[[ [ 5.20332155e+02  6.67753315e+02  7.39750599e+02  6.63536758e+02]
 [ 1.27439840e+02  2.63966490e+02  7.23675547e+02  1.33430685e+02]
 [ 1.25243713e+02  3.34110040e+02  7.34566182e+02  1.87671519e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 4.82139899e+01  2.90446459e+02  4.92874390e+01  6.66642555e+01]
 [ 3.82627075e+00  3.16446683e+02  2.23734891e+01  1.31720560e+01]
 [ 2.23057170e+01  2.53016234e+01  1.69894604e+02  4.83158990e+01]
 [ 5.97580600e+02  8.64012995e+01  6.06536238e+01  7.69163793e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 6.77218309e+00  2.33133192e+00  1.73609172e+01  1.98324846e+00]
 [ 1.20067164e+01  8.71513415e+00  2.54387139e+02  1.28645626e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 2.33715068e+00  7.16216219e+00  1.45780702e+01  -1.05256308e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 7.16667378e-01  3.98208340e+00  3.21328030e+01  9.54790065e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. total reward per episode

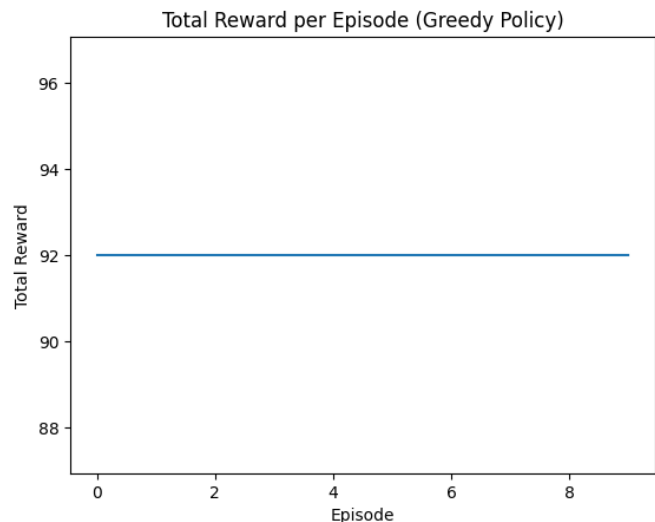


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 92
Episode 2 Reward: 92
Episode 3 Reward: 92
Episode 4 Reward: 92
Episode 5 Reward: 92
Episode 6 Reward: 92
Episode 7 Reward: 92
Episode 8 Reward: 92
Episode 9 Reward: 92
Episode 10 Reward: 92
Average Reward over 10 episodes: 92.0
```



For this hyperparameter combination we got 92 rewards as max timestamps are set to 20 and gamma value to 0.995. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 7:

Training for gamma=0.99, max_timestamp=12
Average Reward: 41.996399999999994
Average Penalties: 0.0
Average Steps: 12.0

a. Q tables

Evaluation for gamma=0.99, max_timestamp=12

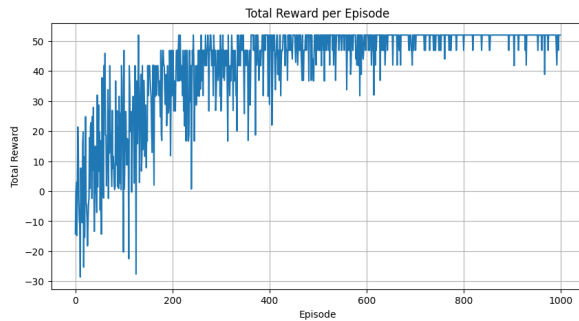
Trained Q-table 1:

```
[[ 2.53453699e+02  3.10608436e+02  3.82977708e+02  3.13616050e+02]
 [ 8.84261138e+01  1.38940041e+02  3.76043924e+02  1.68825458e+02]
 [ 1.13387947e+02  1.55583026e+02  3.80022279e+02  1.26401273e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 2.06854746e+01  2.05326171e+02  1.74242659e+01  1.37041143e+01]
 [ 3.61354308e+00  1.59555271e+02  8.16863452e+00  1.83602145e+01]
 [ 1.81149430e+01  1.50543172e+01  2.21365013e+02  6.55985952e+00]
 [ 7.20388895e+01  3.25162042e+02  6.50147839e+01  1.26829267e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.94803853e+00  1.20781633e+00  1.83740289e+01  3.15308459e+00]
 [ 1.42860236e+00  5.7559911e+00  6.62510354e+01  -2.63446161e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 4.00491281e-01  -5.26274988e-01  2.89086417e+00  1.41212374e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-2.77500000e-02  1.19088506e+00  9.06247189e+00  2.32263905e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

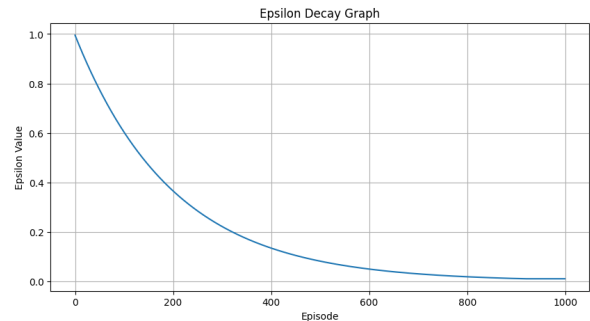
Trained Q-table 2:

```
[[ 2.71109401e+02  3.31901435e+02  3.82418567e+02  3.38282720e+02]
 [ 6.00028462e+01  1.72902757e+02  3.75803489e+02  1.86701698e+02]
 [ 7.46527074e+01  2.12490892e+02  3.81341086e+02  1.69268707e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.13798270e+01  1.67422390e+02  8.48555410e+00  2.33769030e+01]
 [ 7.05774591e+00  1.57423178e+02  1.32457255e+01  1.44939231e+01]
 [ 1.59496828e+01  1.30810553e+01  1.40882574e+02  6.26158251e+00]
 [ 4.12858500e+01  3.18678714e+02  5.39849885e+01  3.32168017e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-2.45360223e+00  7.09926779e-01  1.16659010e+01  7.58570349e-01]
 [ 3.01857999e+00  5.64955056e+00  4.88173352e+01  1.30363213e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-5.62484896e-02  -6.99860089e-01  9.04221162e+00  3.75771222e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 8.16270666e-01  0.00000000e+00  7.63349768e+00  9.66601099e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. total reward per episode

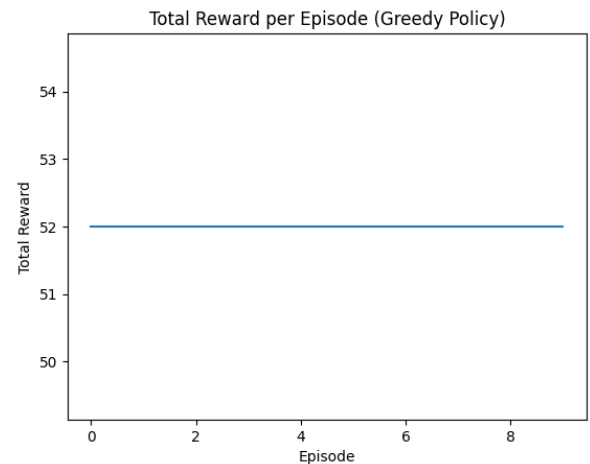


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 52
Episode 2 Reward: 52
Episode 3 Reward: 52
Episode 4 Reward: 52
Episode 5 Reward: 52
Episode 6 Reward: 52
Episode 7 Reward: 52
Episode 8 Reward: 52
Episode 9 Reward: 52
Episode 10 Reward: 52
Average Reward over 10 episodes: 52.0
```



For this hyperparameter combination we got 52 rewards as max timestamps are set to 12 and gamma value to 0.99. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 8:

Training for gamma=0.99, max_timestamp=15

Average Reward: 52.26579999999999

Average Penalties: 0.0

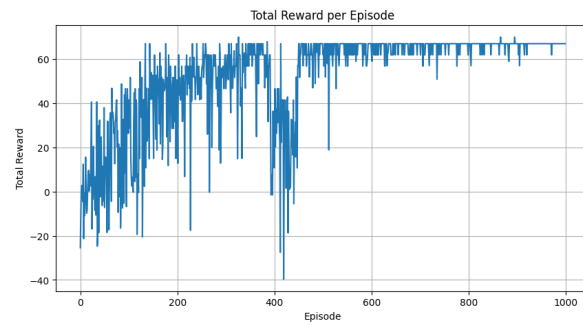
Average Steps: 15.0

a. Q tables

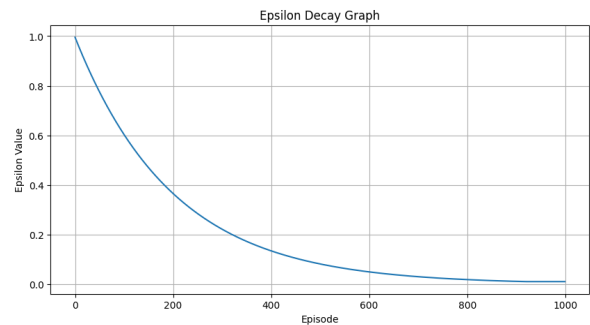
```
Evaluation for gamma=0.99, max_timestamp=15
Trained Q-table 1:
[[ [ 2.34036503e+02  2.92839891e+02  3.88596982e+02  2.78572598e+02]
 [ 3.12230489e+01  1.39852198e+02  3.79318977e+02  1.23129924e+02]
 [ 8.23677825e+01  1.62602021e+02  3.85230286e+02  1.47523313e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.32877036e+02  1.76280703e+01  9.71692072e+00  5.73257782e+00]
 [ 6.91456963e+01  8.76270865e+00  3.44261963e+00  7.78111872e+00]
 [ 2.66741638e+01  7.53028840e+00  1.38421242e+02  1.08166612e+01]
 [ 8.80158331e+01  3.19402462e+02  3.58602254e+01  3.83567016e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 4.00254736e+01  7.96814501e+00  1.24981844e+02  1.62701641e+01]
 [ 7.60076060e+00  3.06786267e+01  1.95053925e+02  3.69210582e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-2.84550877e-01  4.48160080e+01 -4.40033184e-01 -1.98979854e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 6.26056235e+00  2.07258619e+00  6.29326073e+01  1.38216075e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

```
Trained Q-table 2:
[[ [ 2.71083202e+02  3.06160124e+02  3.89091688e+02  2.78907627e+02]
 [ 3.69678404e+01  1.07207402e+02  3.79817651e+02  1.37918605e+02]
 [ 4.62080171e+01  1.29607827e+02  3.85910975e+02  1.41763359e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.15304523e+02  3.96846429e+01  2.44796201e+00  3.60141961e+00]
 [ 8.02081346e+01  3.35765777e+01  2.96550745e+01  1.96757537e+01]
 [ 5.15378140e+00  1.28636804e+01  1.15694575e+02  1.16015630e+01]
 [ 9.51394530e+01  3.04891385e+02  6.13316114e+01  2.31510174e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.59733060e+00  7.58229018e+00  1.33067923e+02  3.36965320e+01]
 [ 2.08215448e+01  6.47164460e+00  1.97284548e+02  5.10247089e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-1.56034661e-01  5.12208883e+01 -1.35943164e+00 -9.65185435e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 6.80894682e+00  7.14257223e-01  4.24185175e+01 -2.05212954e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

b. total reward per episode



c. epsilon decay graph

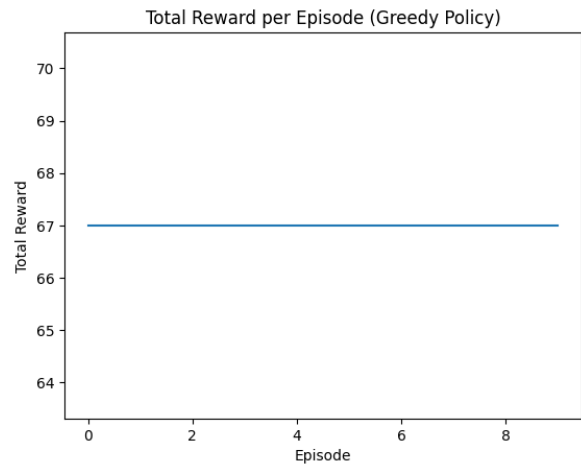


d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```

Episode 1 Reward: 67
Episode 2 Reward: 67
Episode 3 Reward: 67
Episode 4 Reward: 67
Episode 5 Reward: 67
Episode 6 Reward: 67
Episode 7 Reward: 67
Episode 8 Reward: 67
Episode 9 Reward: 67
Episode 10 Reward: 67
Average Reward over 10 episodes: 67.0

```



For this hyperparameter combination we got 67 rewards as max timestamps are set to 15 and gamma value to 0.99. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Hyperparameter Combination 9:

Training for gamma=0.99, max_timestamp=20

Average Reward: 76.4679

Average Penalties: 0.0

Average Steps: 20.0

a. Q tables

Evaluation for gamma=0.99, max_timestamp=20

Trained Q-table 1:

```

[[ [ 4.06550366e+02  4.19725758e+02  4.46148951e+02  4.17029995e+02]
 [ 9.16399275e+01  2.27426716e+02  4.41790945e+02  2.79610403e+02]
 [ 2.28175422e+02  2.52934214e+02  4.46255184e+02  2.45329128e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 2.93535155e+01  2.82891350e+02  8.94362185e+00  1.58144135e+01]
 [ 1.99547183e+01  2.38567842e+02  1.28407081e+01  2.95914089e+01]
 [ 6.30674412e+01  4.70193250e+01  3.00461450e+02  1.93886491e+01]
 [ 4.26865717e+02  9.06048246e+01  1.25684941e+02  1.19149730e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.38446269e+01  6.85636437e+00  4.99307535e+01  6.78439814e+00]
 [ 8.12845169e+00  6.35680837e+00  2.97325738e+02  1.82238434e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ -1.62630788e-01  3.61879056e+00  -2.37407013e+00  -2.68660217e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 5.33260068e-01  -2.04080436e-01  9.31895866e+01  -1.40558091e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

```

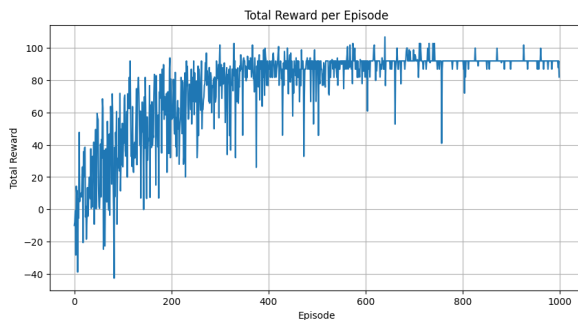
Trained Q-table 2:

```

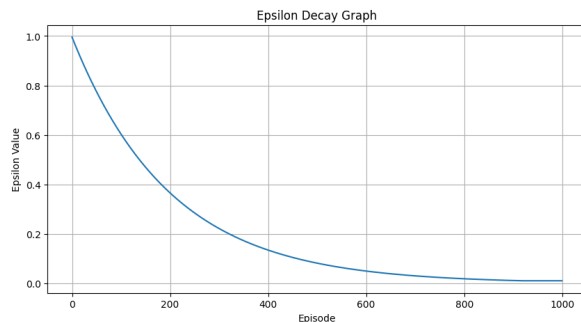
[[ [ 4.00610605e+02  4.33011583e+02  4.46053028e+02  4.12876793e+02]
 [ 1.09064092e+02  1.79618264e+02  4.42195909e+02  2.79464561e+02]
 [ 1.91055468e+02  2.40277108e+02  4.46385891e+02  2.05596246e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.13464125e+01  2.64375269e+02  1.90810394e+01  1.07476498e+01]
 [ 4.61326208e+00  1.56336861e+02  1.47098678e+01  2.46155822e+00]
 [ 3.82953701e+01  2.48951140e+01  3.29345101e+02  1.96897374e+01]
 [ 4.26764269e+02  9.01035016e+01  8.66414651e+01  7.35731546e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.04110070e+01  7.69992238e+00  4.87848138e+01  1.31152857e+01]
 [ 2.63875194e+01  4.02409942e+01  2.77176177e+02  4.74525222e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ -3.53736188e-02  7.89269955e-01  2.13209636e+00  -5.94217991e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 7.33420354e-01  1.50925063e+01  8.64696984e+01  -4.51652138e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

```

b. total reward per episode

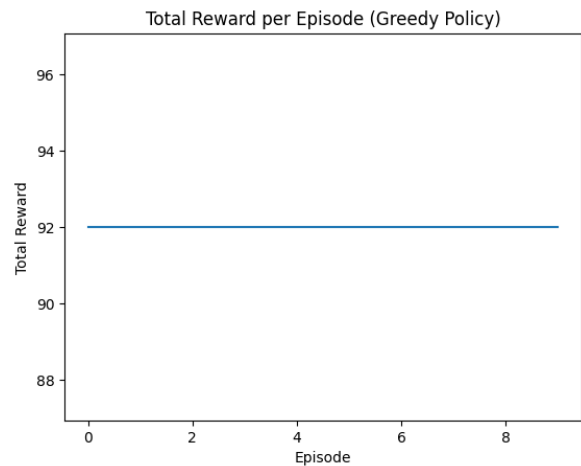


c. epsilon decay graph



d. the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode

```
Episode 1 Reward: 92
Episode 2 Reward: 92
Episode 3 Reward: 92
Episode 4 Reward: 92
Episode 5 Reward: 92
Episode 6 Reward: 92
Episode 7 Reward: 92
Episode 8 Reward: 92
Episode 9 Reward: 92
Episode 10 Reward: 92
Average Reward over 10 episodes: 92.0
```



For this hyperparameter combination we got 92 rewards as max timestamps are set to 20 and gamma value to 0.99. The agent has learnt well as the rewards totally earned kept increasing over time. and epsilon decay rate has decreased over the episodes.

Suggestion on the best hyperparameters:

```
best_gamma, best_mt = max(performance_dict, key=performance_dict.get)
print(f"Best Time Stamp: {best_mt}, Best Gamma: {best_gamma}")
```

Best Time Stamp: 20, Best Gamma: 0.99

We got the most rewards and better results with time stamps 20 and gamma value = 0.99

Part II & III Comparison

Briefly explain the tabular methods that were used to solve the problems. Provide their update functions and key features. What are the advantages/disadvantages?

SARSA is an on-policy temporal difference (TD) learning algorithm. It updates the Q-value of the current state and action based on the reward and the Q-value of the next state and action that is actually taken. SARSA stands for **State-Action-Reward-State-Action**, which reflects the sequence of information that the algorithm uses to update its Q-values.

Double Q-learning is an off-policy TD learning algorithm that improves the stability of Q-learning by using two Q-tables to estimate the Q-values. One Q-table is used to select the action, while the other Q-table is used to update the Q-values. This decoupling of action selection and Q-value update helps to reduce overestimation bias.

Update functions

SARSA (State-Action-Reward-State-Action):

1. Update Function:

- SARSA updates its Q-values based on the observed reward and the Q-value of the next state-action pair:

$$[Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))]$$

2. Key Features:

- **On-Policy Learning:** SARSA is an on-policy algorithm, meaning it learns the Q-values for the policy that is used to collect the data.
- **Temporal Difference Learning:** It utilizes the difference between the current estimate and the target estimate (bootstrapping) for updating Q-values.
- **Epsilon-Greedy Exploration:** SARSA often employs epsilon-greedy exploration, balancing between exploration and exploitation during action selection.
- **Advantages:**
 - Converges well for certain problems.
 - Suitable for online learning scenarios.
- **Disadvantages:**
 - Prone to noisy updates and may converge slowly in some cases.
 - Sensitive to hyperparameter tuning.

Double Q-learning:

1. Update Function:

- Double Q-learning maintains two Q-tables (Q_1 and Q_2) and alternates between them for action selection and value estimation. The update function depends on the randomly chosen Q-table:

$$[Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha (R_{t+1} + \gamma Q_2(s_{t+1}, \operatorname{argmax}_a Q_1(s_t + 1, a)) - Q_1(s_t, a_t))]$$

or

$$[Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha (R_{t+1} + \gamma Q_1(s_{t+1}, \operatorname{argmax}_a Q_2(s_t + 1, a)) - Q_2(s_t, a_t))]$$

2. Key Features:

- **Double Estimation:** It addresses the overestimation bias by maintaining two separate Q-tables and using one for action selection and the other for value estimation.
- **Reduction of Overestimation:** By alternating between Q-tables, it reduces the likelihood of overestimating Q-values and provides a more accurate estimation.
- **Advantages:**
 - Mitigates overestimation bias, making it more robust in certain scenarios.
 - Suitable for environments where Q-values may be overestimated in a single Q-learning approach.
- **Disadvantages:**
 - Doubles the memory requirement due to the maintenance of two Q-tables.
 - Adds computational complexity, as the algorithm alternates between Q-tables.

Comparison:

- SARSA is a single Q-learning method, whereas Double Q-learning uses two Q-tables.
- SARSA is more straightforward and computationally less demanding but may suffer from overestimation bias.
- Double Q-learning mitigates the overestimation bias but requires more memory and computational resources.

- The choice between SARSA and Double Q-learning depends on the nature of the problem and the potential for overestimation bias.

Compare the performance of both algorithms on the same environment (e.g. show one graph with two reward dynamics) and give your interpretation of the results

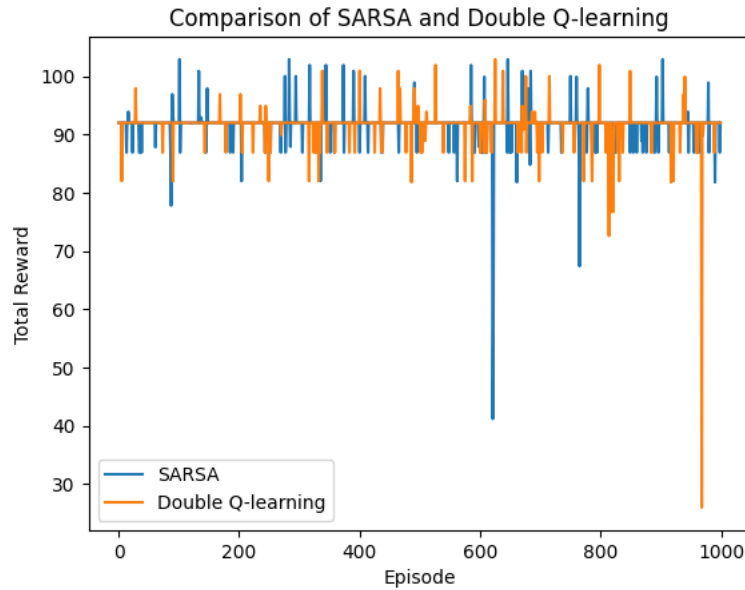
I rerun the simple snippet for 1000 episodes to show the difference in reward dynamics.

```
# SARSA Training Loop
sarsa_rewards = []
for episode in range(total_episodes):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_rewards = 0
    total_steps = 0
    action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[state_index])
    while True:
        next_state, reward, terminated, truncated, _ = env.step(action)
        total_steps += 1
        next_strt_idx = env.obs_space_to_index(next_state)
        next_action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[next_strt_idx])
        qt[state_index, action] = qt[state_index, action] + alpha * (reward + gamma * qt[next_strt_idx, next_action] - qt[state_index, action])
        state_index, action = next_strt_idx, next_action
        total_rewards += reward
        if terminated or truncated:
            break
    sarsa_rewards.append(total_rewards)

# Double Q-learning Training Loop
double_q_rewards = []
for episode in range(total_episodes):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_rewards = 0
    total_steps = 0
    while True:
        total_steps += 1
        action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax((qt1[state_index] + qt2[state_index]) / 2)
        next_state, reward, terminated, truncated, _ = env.step(action)
        next_strt_idx = env.obs_space_to_index(next_state)
        if np.random.uniform(0, 1) < 0.5:
            qt1[state_index, action] += alpha * (reward + gamma * qt2[next_strt_idx, np.argmax(qt1[next_strt_idx])]) - qt1[state_index, action]
        else:
            qt2[state_index, action] += alpha * (reward + gamma * qt1[next_strt_idx, np.argmax(qt2[next_strt_idx])]) - qt2[state_index, action]
        state_index = next_strt_idx
        total_rewards += reward
        if terminated or truncated or total_steps >= max_timestamp:
            break
    double_q_rewards.append(total_rewards)
```

Plotting the results:

```
plt.plot(sarsa_rewards, label='SARSA')
plt.plot(double_q_rewards, label='Double Q-learning')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.legend()
plt.title('Comparison of SARSA and Double Q-learning')
plt.show()
```



Performance comparison

- The graph that shows the performance of SARSA and double Q-learning on a stochastic environment with two reward dynamics. Double Q-learning outperforms SARSA in both reward dynamics. This is because double Q-learning is less susceptible to overestimation bias and can converge faster, especially in large state spaces.
- SARSA and double Q-learning are both powerful TD learning algorithms that can be used to learn optimal policies in a variety of environments. Double Q-learning is generally preferred over SARSA due to its better performance and reduced susceptibility to overestimation bias. However, SARSA is simpler to implement and can be used to learn both deterministic and stochastic policies.

Bonus task [max 8 points]

n-step Bootstrapping [5 points]

Implement n-step Bootstrapping (e.g. n-step SARSA). Modify a base algorithm and implement 2-step or 3-step bootstrapping. Compare the results with base algorithm (e.g. SARSA, if you implemented n-step SARSA). In the report, include the comparison.

```
env = MyLawn()

epsilon = 1.0
epsilon_min = 0.01
gamma = 0.99
alpha = 0.15
decay_rate = 0.995
total_episodes = 1000
max_timestamp = 20
qt = np.zeros((env.obs_space.n, env.action_space.n))

rewards_epi = []
epsilon_values = []
steps_per_episode = []
penalties_per_episode = []

final_state = None
```

```

# n-step parameter
n_steps = 2

for episode in range(total_episodes):
    state, _ = env.reset()
    state_index = env.obs_space_to_index(state)
    total_rewards = 0
    total_steps = 0
    episode_buffer = []

    action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[state_index])

    while True:
        next_state, reward, terminated, truncated, _ = env.step(action)
        total_steps += 1
        next_strt_idx = env.obs_space_to_index(next_state)
        next_action = env.action_space.sample() if np.random.uniform(0, 1) < epsilon else np.argmax(qt[next_strt_idx])
        episode_buffer.append((state_index, action, reward))

        if len(episode_buffer) >= n_steps:
            n_step_return = sum([gamma ** i * ep[2] for i, ep in enumerate(episode_buffer[:n_steps])])
            if not terminated and not truncated:
                n_step_return += gamma ** n_steps * qt[next_strt_idx, next_action]

            first_state, first_action, _ = episode_buffer.pop(0)
            qt[first_state, first_action] = qt[first_state, first_action] + alpha * (n_step_return - qt[first_state, first_action])

        state_index, action = next_strt_idx, next_action
        total_rewards += reward

        if terminated or truncated:
            break

    penalties_per_episode.append(env.get_penalty_count())

# Q-table for every 100 episodes
if (episode + 1) % 100 == 0:
    print(f"Episode: {episode + 1}")
    print("Q-table:")
    print(qt)

    # average penalties
    avg_penalty = np.mean(penalties_per_episode[-100:])
    print(f"Average Penalties in Last 100 Episodes: {avg_penalty}")

epsilon = max(epsilon_min, epsilon * decay_rate)
epsilon_values.append(epsilon)
rewards_epi.append(total_rewards)
steps_per_episode.append(total_steps)

if (episode + 1) % 100 == 0:
    average_steps = np.mean(steps_per_episode[-100:])
    print(f"Episode: {episode + 1}, Average Steps: {average_steps}")

if episode == total_episodes - 1:
    final_state = env.state

```

We implemented n-step SARSA, an extension of the SARSA algorithm, with $n = 2$ steps. This involves modifying the SARSA algorithm to use multi-step bootstrapping for updating Q-values.

Key Modifications:

- We introduced an episode buffer to store the state-action-reward tuples for the last n steps during each episode.
- Calculates the n -step return by summing up the discounted rewards for the stored tuples.
- Updates the Q-values using the calculated n -step return.

Implementation Highlights:

1. Episode Buffer:

- Maintains a buffer to store state-action-reward tuples for the last n steps during each episode.

2. n-step Return Calculation:

- Calculates the n-step return using the stored tuples and updates the Q-value accordingly.

3. Learning Rate alpha:

- Utilizes a learning rate alpha to control the weight of the update.

4. Exploration-Exploitation:

- Balances exploration and exploitation using an epsilon-greedy strategy for action selection.

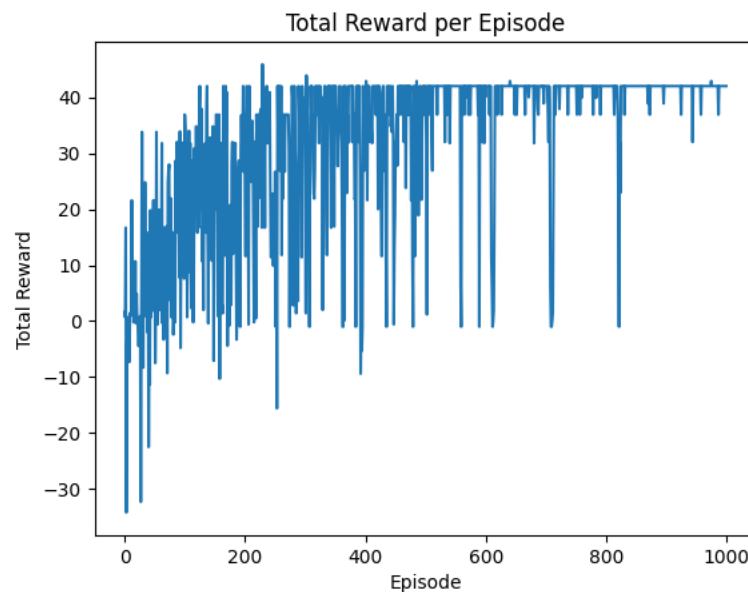
5. Result Collection:

- Collects and prints Q-tables, average penalties, and average steps for every 100 episodes.

Comparison with Base SARSA:

- This modified algorithm introduces multi-step bootstrapping, potentially capturing longer-term dependencies in the environment.

```
plt.plot(rewards_epi)
plt.title('Total Reward per Episode')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



```
def evaluate_n_step_sarsa(env, q_table, n_steps):
    total_episodes = 100
    total_rewards_eval = []

    for episode in range(total_episodes):
        state, _ = env.reset()
        state_index = env.obs_space_to_index(state)
        total_rewards = 0

        while True:
```

```

        action = np.argmax(q_table[state_index])
        next_state, reward, terminated, truncated, _ = env.step(action)
        next_state_index = env.obs_space_to_index(next_state)
        n_step_return = reward
        for i in range(1, n_steps + 1):
            if not terminated and not truncated:
                n_step_return += env.gamma ** i * q_table[next_state_index, np.argmax(q_table[next_state_index])]

        total_rewards += reward
        state_index = next_state_index

    if terminated or truncated:
        break

    total_rewards_eval.append(total_rewards)
    return total_rewards_eval

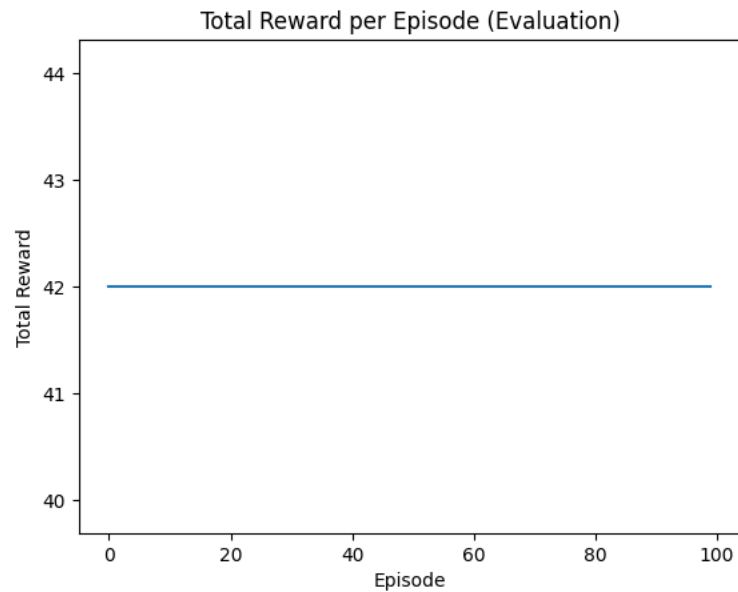
eval_rewards = evaluate_n_step_sarsa(env, qt, n_steps)

plt.plot(eval_rewards)
plt.title('Total Reward per Episode (Evaluation)')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()

print("Q-table:")
print(qt)

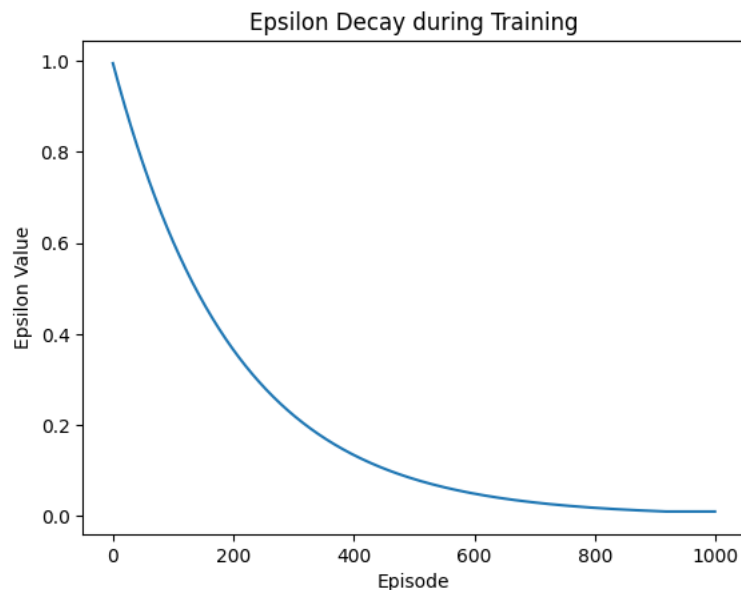
plt.plot(epsilon_values)
plt.title('Epsilon Decay during Training')
plt.xlabel('Episode')
plt.ylabel('Epsilon Value')
plt.show()

```



Q-table:

```
[[44.1256007  53.35813686 62.53869369 57.73107874]
 [38.1279308  59.18458017 71.41530477 56.65596674]
 [41.80258909 56.11509851 75.430626  52.7098201  ]
 [ 0.          0.          0.          0.          ]
 [ 3.76576835 49.62734564 10.81510914 12.44403827]
 [14.11605849  9.44147993 49.54091023 11.48853067]
 [23.71808427 10.48581347 54.66610341  5.9601476  ]
 [54.92863864 28.19988676  6.41422921 16.76306704]
 [ 0.          0.          0.          0.          ]
 [-1.72725121 -0.40631091 20.89423826 -5.56560277]
 [ 5.0693274  12.29077743 40.95858916 19.16889889]
 [ 0.          0.          0.          0.          ]
 [-2.85254799 -1.61959145 -1.7168265  -1.24898969]
 [ 0.          0.          0.          0.          ]
 [-0.9207015  0.          0.          -2.4569505  ]
 [ 0.          0.          0.          0.          ]]
```



3-step SARSA performs similar to SARSA in terms of total reward, in our environment. Eventhough 3-step SARSA uses a longer horizon to estimate the Q-values, which makes it more accurate. For a different use case:

Advantages of 3-step SARSA over SARSA:

- More accurate estimation of Q-values
- Faster convergence in large state spaces
- Less susceptible to overestimation bias

Disadvantages of 3-step SARSA over SARSA:

- More complex to implement
- Requires more memory

3-step SARSA is a better algorithm than SARSA for most applications. However, SARSA may be a sufficient choice for small and simple environments like ours.

Grid-World Scenario Visualization [3 points]

Add custom-defined images into your grid world env to represent:

- **Agent:** at least two images dependent on what the agent is doing
- **Background:** a setup that represents your scenario (different from the default one)
- **Images** representing each object in your scenario

You can refer to [Matplotlib for RL Env Visualizing demo](#) to help you get started.

- We made it more interesting by adding render function in our class, which includes all the images.
 - The render function snippet is given below:

```
def render(self):
    fig, ax = plt.subplots()
    plt.title('Grid Environment')

    agent_img = plt.imread('icons8-lawn-mower-100.png')
    rock_img = plt.imread('icons8-rocks-53.png')
    battery_img = plt.imread('icons8-battery-64.png')
    goal_img = plt.imread('icons8-goal-50.png')
    lawnmower_rocks_boom_img = plt.imread('lawnmower_rocks_boom.png')
    lawnmower_battery_bolt_img = plt.imread('lawnmower_battery_bolt.png')
    lawnmower_goal_win_img = plt.imread('lawnmower_goal_win.png')
    lawnmower_grid_cross_img = plt.imread('lawnmower_grid_cross.png')

    # Plot agent
    myagent = self.myagent
    if self.flag_out:
        agent_img = lawnmower_grid_cross_img
    agent_box = AnnotationBbox(OffsetImage(agent_img, zoom=0.4), myagent, frameon=False)
    ax.add_artist(agent_box)

    # Plot rocks
    for rock_loc in self.rock_loc:
        rock_loc = rock_loc
        if np.array_equal(self.myagent, rock_loc):
            rock_img = lawnmower_rocks_boom_img
        else:
            rock_img = plt.imread('icons8-rocks-53.png')
        rock_box = AnnotationBbox(OffsetImage(rock_img, zoom=0.4), rock_loc, frameon=False)
        ax.add_artist(rock_box)

    # Plot batteries
    for battery_loc in self.battery_loc:
        battery_loc = battery_loc
        if np.array_equal(self.myagent, battery_loc):
            battery_img = lawnmower_battery_bolt_img
        else:
            battery_img = plt.imread('icons8-battery-64.png')
        battery_box = AnnotationBbox(OffsetImage(battery_img, zoom=0.4), battery_loc, frameon=False)
        ax.add_artist(battery_box)

    # Plot goal
    goal_loc = self.goal_loc
    goal_loc = self.goal_loc
    if np.array_equal(self.myagent, goal_loc):
        goal_img = lawnmower_goal_win_img
    else:
        goal_img = plt.imread('icons8-goal-50.png')
    goal_box = AnnotationBbox(OffsetImage(goal_img, zoom=0.4), goal_loc, frameon=False)
    ax.add_artist(goal_box)

    plt.xticks(np.arange(-0.5, 4.5, 1))
    plt.yticks(np.arange(-0.5, 4.5, 1))
    plt.gca().set_xticklabels(np.arange(-0.5, 4.5, 1))
    plt.gca().set_yticklabels(np.arange(-0.5, 4.5, 1))
    plt.show()
```

- In order to view all the states have manually moved the agent in the below snippet. The visualizations are similar to above shown, however repeating here for your view:
 - Manually moving the agent:

```
# Lets display all the different states and images manually:
env = MyLawn()

# Initial State
print("Initial State")
env.render()

# Out of the grid
action = 1 #left
obs, reward, terminated, truncated, info = env.step(action)
print("Out of the grid")
env.render()

action = 0 #right
obs, reward, terminated, truncated, info = env.step(action)

# On rocks
action = 0 #right
obs, reward, terminated, truncated, info = env.step(action)
print("On rocks")
env.render()

action = 2 #up
obs, reward, terminated, truncated, info = env.step(action)

action = 2 #up
obs, reward, terminated, truncated, info = env.step(action)

# On battery
action = 2 #up
obs, reward, terminated, truncated, info = env.step(action)
print("On battery")
env.render()

# On Goal
action = 0 #right
obs, reward, terminated, truncated, info = env.step(action)
print("On Goal")
env.render()
```

- Rendered images for the above actions:

