ESE 559 - Learning & Planning in Robotics

Seif Elkhashab / 03-03-2024

Problem 1: Consider a robot residing in the grid world environment of Fig.1.

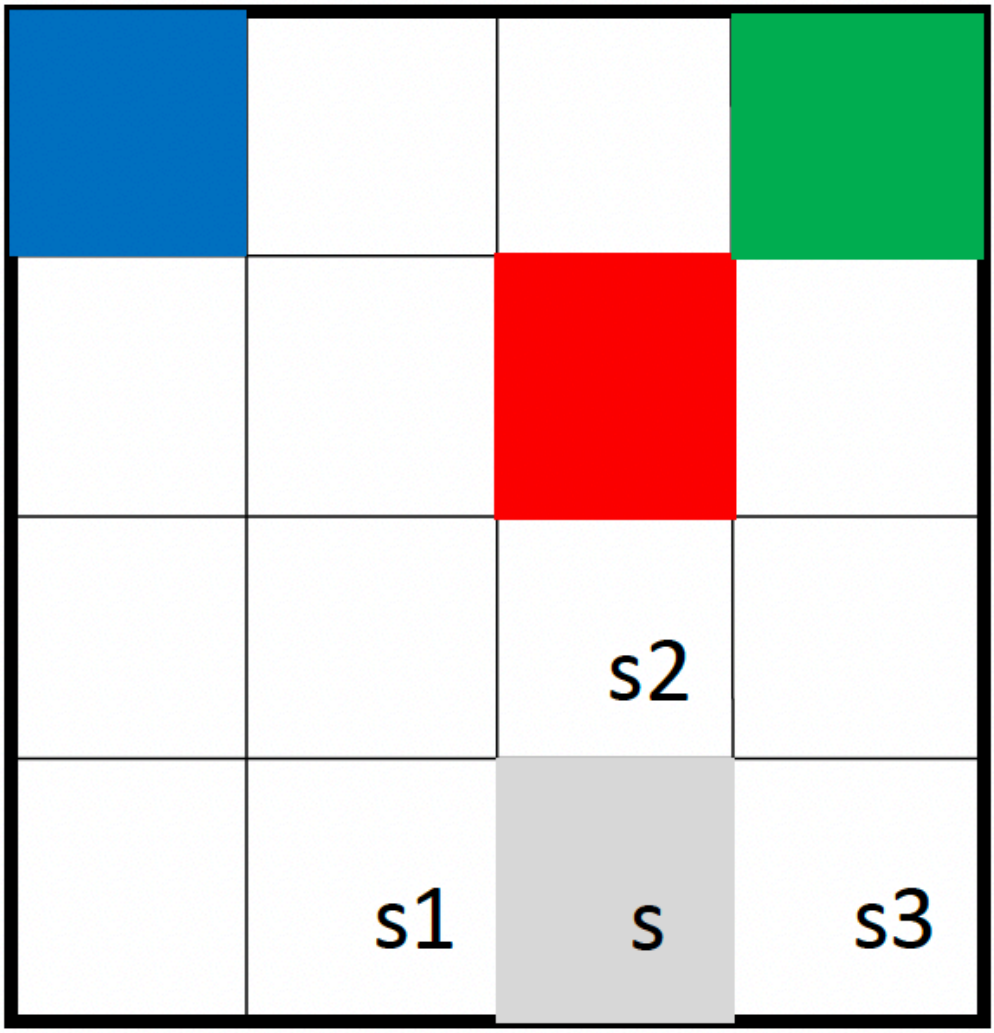


Figure 1: 4x4 grid world

The actions of ther obot are A = {up, down, left, right, idle}. The task of the robot is to eventually reach either the green or the blue cell with the smallest number of steps, while avoiding obstacles (red). The robot does not need to stay in the green/blue region once it arrives there. In other words, what happens after the robot reaches these regions is irrelevant. Therefore, you can treat the blue, green, and red cells as terminal MDP

states. For simplicity, assume that the available actions at each state are only the ones that can keep the robot inside the workspace. When the robot chooses the action “idle” it stays with probability 1 in its current state s. For all other actions, the probability of going to the correct state is 0.7 and the probability of going to a neighboring state is 0.3/m(s) where m(s) is the number of neighboring states of the current state s. For instance, for the state s, shaded with gray color, we have that (i) the only available actions are {up, left, right, idle} and (ii) m(s)=3, P(s, `left', s1)=0.7, P(s, `left', s3)=0.15, and P(s, `left', s2)=0.15.

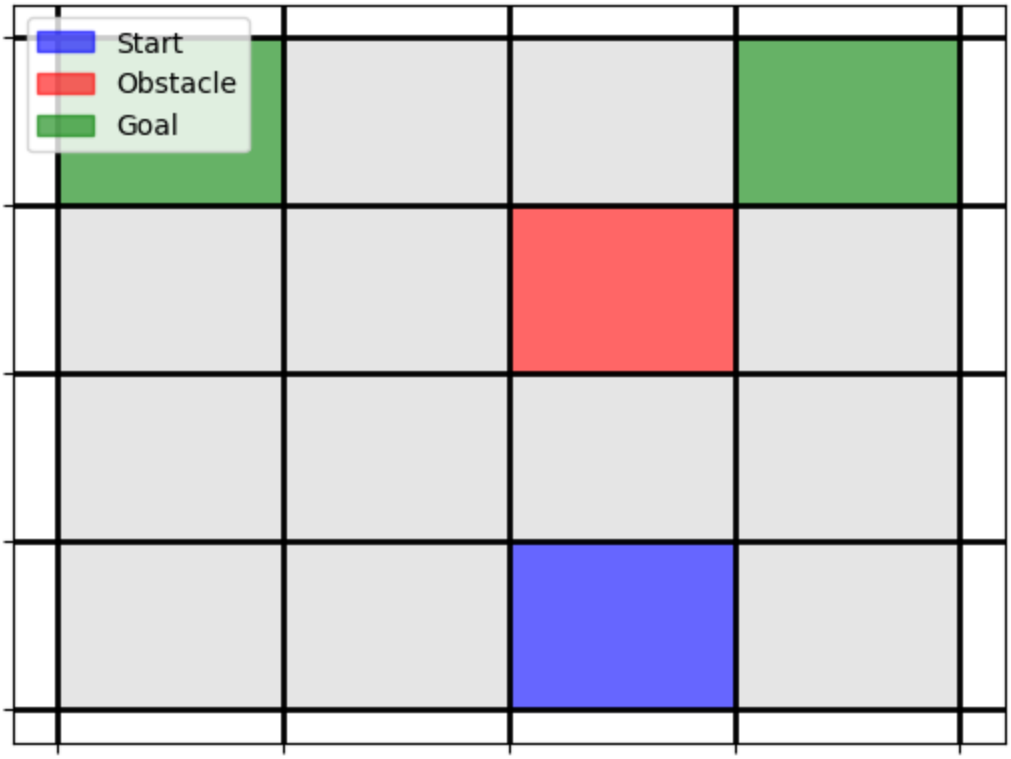
Importing necessary libraries

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
import time
import random
from collections import defaultdict
```

Visualize the Environment

In [ ]:
```
# Initialize the 4x4 grid environment
grid_size = 4
environment = np.zeros((grid_size, grid_size))

# Function to visualize the environment
def visualize_environment(environment, start_pos, goal_positions, obstacles)
    fig, ax = plt.subplots()

    # Create a color map for background, start, goal, and obstacles
    cmap = plt.cm.coolwarm
    norm = plt.Normalize(vmin=-1.5, vmax=1.5)
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])

    # Fill the grid
    for i in range(grid_size):
        for j in range(grid_size):
            if (i, j) == start_pos:
                ax.fill_between([j, j+1], i, i+1, color='blue', label='Start
            elif (i, j) in goal_positions:
                ax.fill_between([j, j+1], i, i+1, color='green', label='Goal
            elif (i, j) in obstacles:
                ax.fill_between([j, j+1], i, i+1, color='red', label='Obstac
            else:
                ax.fill_between([j, j+1], i, i+1, color='grey', alpha=0.2)

    plt.xticks(range(grid_size+1))
    plt.yticks(range(grid_size+1))
    ax.set_xticklabels([])
```

```
    ax.set_yticklabels([])
    plt.grid(which='major', axis='both', linestyle='-', color='k', linewidth
    handles, labels = ax.get_legend_handles_labels()
    unique = [(h, l) for i, (h, l) in enumerate(zip(handles, labels)) if l r
    plt.legend(*zip(*unique), loc='upper left')
    plt.show()

# Defining the starting position, goals, and obstacles
start_pos = (0, 2)
goal_pos = [(3,0), (3, 3)]
obstacles = [(2, 2)]

visualize_environment(environment, start_pos, goal_pos, obstacles)
```



Designing rewards function for the considered task

I implemented a very simple reward function, if the state reaches the goal, 100 points if an obstacle, then -100, and if it doesn't move then -1 so there's incentive for the robot to keep going towards the goal.

In [ ]:
```
goal_states = [(0, 0), (0, 3)]
obstacle_states = [(1, 2)]
start_state = (2, 0)

def get_reward(state, goal_states=goal_states, obstacle_states=obstacle_stat
```

```
    if state in goal_states:
        return 100  # Reward for reaching a goal
    elif state in obstacle_states:
        return -100  # Penalty for hitting an obstacle
    else:
        return -1  # penalty for not making a move

reward = get_reward(start_state, goal_states, obstacle_states)
print(f"Reward: {reward}")
```

```
Reward: -1
```

Implementing Value Iteration

For each state s, figuere out the expected reward of starting in s and acting optimally.

Using the Bellman Equation

The value function near high-reward states will be large

The discounting factor decreases overall value function

In [ ]:
```
# Environment setup
grid_size = 4
goal_states = [(0, 0), (0, 3)]
start_state = (3, 2)
obstacle_states = [(1, 2)]


# Possible actions
actions = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1),
action_prob_success = 0.7

# Initialize state values
V = np.zeros((grid_size, grid_size))

# Discount factor
gamma = 0.99

# Value iteration function
def value_iteration(V, theta=0.001, max_iterations=1000):
    for iteration in range(max_iterations):
        delta = 0
        for i in range(grid_size):
            for j in range(grid_size):
                if (i, j) in goal_states or (i, j) in obstacle_states:
                    continue
                v = V[i, j]
                V[i, j] = max([compute_state_value((i, j), a, V) for a in ac
                delta = max(delta, abs(v - V[i, j]))
        if delta < theta:
            break
    return V, iteration + 1
```

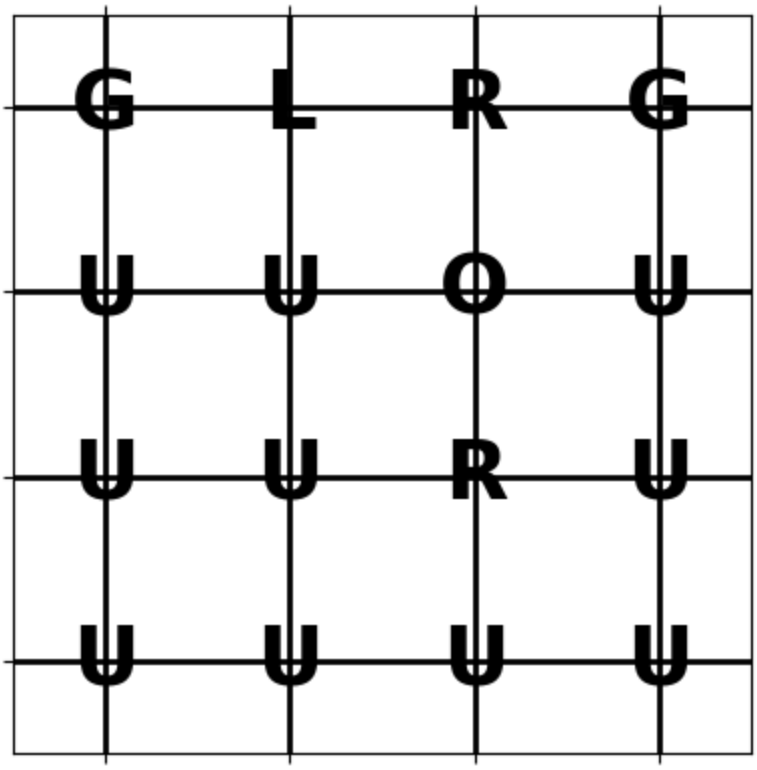
```
# Compute state value for a given action
def compute_state_value(state, action, V):
    if action == 'idle':
        return get_reward(state) + gamma * V[state]
    else:
        transitions = get_transitions(state, action)
        value = 0
        for prob, next_state in transitions:
            reward = get_reward(next_state)
            value += prob * (reward + gamma * V[next_state])
        return value

# Get transitions for a given action
def get_transitions(state, action):
    base_action = actions[action]
    base_next_state = (state[0] + base_action[0], state[1] + base_action[1])

    transitions = []  # List of (probability, next_state)
    if base_next_state not in obstacle_states and is_valid_state(base_next_s
        transitions.append((action_prob_success, base_next_state))
    else:
        transitions.append((action_prob_success, state))  # Bump into wall c

    # Adding neighboring states due to slip
    for act in actions:
        if act == action or act == 'idle':
            continue
        next_state = (state[0] + actions[act][0], state[1] + actions[act][1]
        if next_state not in obstacle_states and is_valid_state(next_state):
            transitions.append((0.3 / len(actions), next_state))
        else:
            transitions.append((0.3 / len(actions), state))  # Bump or stay

    return transitions

# Check if state is within grid bounds
def is_valid_state(state):
    return 0 <= state[0] < grid_size and 0 <= state[1] < grid_size

# Perform value iteration
V_final, iterations = value_iteration(V)

print(f"Value function after {iterations} iterations:")
print(V_final)
```

```
Value function after 11 iterations:
[[ 0.         84.00016959 84.89515395  0.        ]
 [82.88610037 69.82824689  0.         83.80477043]
 [67.2542621  57.65874074 55.61733328 67.82463769]
 [55.12525762 47.9676369  46.57402744 55.47995216]]
```

Visualizing the Optimal Decision at every state

In [ ]:
```
# Finding the optimal policy
def get_optimal_policy(V):
    policy = np.full((grid_size, grid_size), ' ', dtype='<U5')
    for i in range(grid_size):
        for j in range(grid_size):
            if (i, j) in goal_states or (i, j) in obstacle_states:
                policy[i, j] = 'G' if (i, j) in goal_states else 'O'
                continue
            best_action = None
            best_value = float('-inf')
            for action in actions:
                value = compute_state_value((i, j), action, V)
                if value > best_value:
                    best_value = value
                    best_action = action
            policy[i, j] = best_action[0].upper() if best_action != 'idle' e
    return policy

optimal_policy = get_optimal_policy(V_final)
print("Optimal Policy (U=Up, D=Down, L=Left, R=Right, I=Idle, G=Goal, X=Obst
print(optimal_policy)

# Visualizeing the policy
def visualize_policy(policy):
    fig, ax = plt.subplots()
    ax.matshow(np.zeros(policy.shape), cmap='Greys', alpha=0.2)

    for i in range(policy.shape[0]):
        for j in range(policy.shape[1]):
            ax.text(j, i, policy[i, j], va='center', ha='center', fontweight

    plt.xticks(range(grid_size))
    plt.yticks(range(grid_size))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.grid(which='major', axis='both', linestyle='-', color='k', linewidth
    plt.show()

visualize_policy(optimal_policy)
```

```
Optimal Policy (U=Up, D=Down, L=Left, R=Right, I=Idle, G=Goal, X=Obstacle):
[['G' 'L' 'R' 'G']
 ['U' 'U' 'O' 'U']
 ['U' 'U' 'R' 'U']
 ['U' 'U' 'U' 'U']]
```

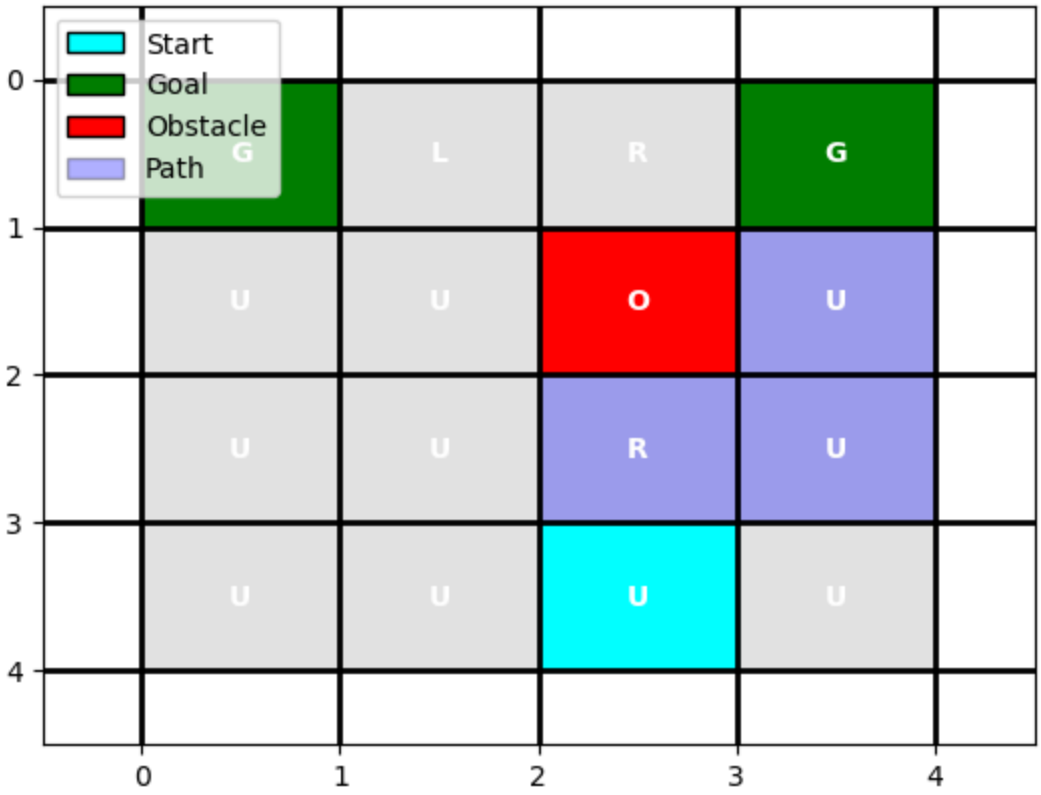


Visualizing a path from starting position

In [ ]:
```
# Defining the visualize_policy_with_path function
def visualize_policy_with_path(policy, path, start_state, goal_states, obsta
    fig, ax = plt.subplots()
    for i in range(policy.shape[0]):
        for j in range(policy.shape[1]):
            rect = plt.Rectangle((j, i), 1, 1, color='grey', alpha=0.2)
            ax.add_patch(rect)

    # Highlighting the path
    for state in path:
        rect = plt.Rectangle((state[1], policy.shape[0] - state[0] - 1), 1,
        ax.add_patch(rect)

    # Coloring in the start, goal, and obstacle squares
    start_rect = plt.Rectangle((start_state[1], policy.shape[0] - start_stat
    ax.add_patch(start_rect)
    for goal in goal_states:
        goal_rect = plt.Rectangle((goal[1], policy.shape[0] - goal[0] - 1),
        ax.add_patch(goal_rect)
    for obstacle in obstacle_states:
        obstacle_rect = plt.Rectangle((obstacle[1], policy.shape[0] - obstac
        ax.add_patch(obstacle_rect)

    # Drawing the policy directions
    for i in range(policy.shape[0]):
        for j in range(policy.shape[1]):
            ax.text(j + 0.5, policy.shape[0] - i - 0.5, policy[i, j], va='ce
```

```
    plt.xlim(0, 4)
    plt.ylim(0, 4)
    plt.xticks(np.arange(5), labels=np.arange(5))
    plt.yticks(np.arange(5), labels=np.arange(4, -1, -1))
    ax.grid(which='major', color='k', linestyle='-', linewidth=2)
    ax.set_xticks(np.arange(6) - .5, minor=True)
    ax.set_yticks(np.arange(6) - .5, minor=True)
    ax.tick_params(which="minor", size=0)

    # Adding a legend
    legend_elements = [
        Patch(facecolor='cyan', edgecolor='k', label='Start'),
        Patch(facecolor='green', edgecolor='k', label='Goal'),
        Patch(facecolor='red', edgecolor='k', label='Obstacle'),
        Patch(facecolor='blue', edgecolor='k', alpha=0.3, label='Path')
    ]
    ax.legend(handles=legend_elements, loc='upper left')

    plt.show()


def trace_path(start_state, policy):
    current_state = start_state
    path = [current_state]

    while True:
        action = policy[current_state[0], current_state[1]]
        if action == 'G' or action == 'O':
            break

        # Determining the next state based on the action
        if action == 'U':
            next_state = (current_state[0] - 1, current_state[1])
        elif action == 'D':
            next_state = (current_state[0] + 1, current_state[1])
        elif action == 'L':
            next_state = (current_state[0], current_state[1] - 1)
        elif action == 'R':
            next_state = (current_state[0], current_state[1] + 1)
        elif action == 'I':
            next_state = current_state

        path.append(next_state)
        current_state = next_state

        if next_state in goal_states:
            break

    return path


# Making a path for visualization
path = trace_path(start_state, optimal_policy)

# Choosing the start rate for experimentation
```

```
start_state = (3, 2)
visualize_policy_with_path(optimal_policy, path, start_state, goal_states, c
```



Implementing Policy Iteration

In [ ]:

```
# Environment setup
grid_size = 4
goal_states = [(0, 0), (0, 3)]
obstacle_states = [(1, 2)]
actions = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1),
action_prob_success = 0.7

 # Discount factor
gamma = 0.99


def is_valid_state(state):
    return 0 <= state[0] < grid_size and 0 <= state[1] < grid_size and state

# Get transitions for a given action
def get_transitions(state, action):
    if action == 'idle':
        return [(1.0, state)]  # Staying in the same state
    next_state = (state[0] + actions[action][0], state[1] + actions[action][
    transitions = []
    if is_valid_state(next_state):
        transitions.append((action_prob_success, next_state))
        slip_probability = (1 - action_prob_success) / (len(actions) - 1)
        for a in actions:
```

```
            if a != action and a != 'idle':
                slip_state = (state[0] + actions[a][0], state[1] + actions[a
                if is_valid_state(slip_state):
                    transitions.append((slip_probability, slip_state))
                else:
                    transitions.append((slip_probability, state))
    else:
        transitions.append((1.0, state))
    return transitions

# Policy evaluation function
def policy_evaluation(policy, V, theta=0.01):
    while True:
        delta = 0
        for i in range(grid_size):
            for j in range(grid_size):
                v = V[i, j]
                new_v = 0
                for action, action_prob in policy[(i, j)].items():
                    for prob, next_state in get_transitions((i, j), action):
                        new_v += action_prob * prob * (get_reward(next_state
                V[i, j] = new_v
                delta = max(delta, abs(v - V[i, j]))
        if delta < theta:
            break
    return V

# Policy improvement function
def policy_improvement(V):
    policy_stable = True
    new_policy = {}
    for i in range(grid_size):
        for j in range(grid_size):
            old_action = max(policy[(i, j)], key=policy[(i, j)].get)
            action_values = {}
            for action in actions:
                action_value = 0
                for prob, next_state in get_transitions((i, j), action):
                    action_value += prob * (get_reward(next_state) + gamma *
                action_values[action] = action_value
            best_action = max(action_values, key=action_values.get)
            new_policy[(i, j)] = {action: 1 if action == best_action else 0
            if old_action != best_action:
                policy_stable = False
    return new_policy, policy_stable

# Initialize policy arbitrarily
policy = {(i, j): {action: 1/len(actions) for action in actions} for i in ra
V = np.zeros((grid_size, grid_size))

# Policy iteration process
iteration = 0
while True:
    V = policy_evaluation(policy, V)
    policy, policy_stable = policy_improvement(V)
    iteration += 1
```

```
    if policy_stable:
        break

# Extract the policy to visualize it
optimal_policy = np.full((grid_size, grid_size), ' ', dtype='<U5')
for i in range(grid_size):
    for j in range(grid_size):
        best_action = max(policy[(i, j)], key=policy[(i, j)].get)
        optimal_policy[i, j] = best_action[0].upper()  # Use the first lette

optimal_policy, iteration


# Making sure the goals and obstacles are marked
for goal_state in goal_states:
    optimal_policy[goal_state[0], goal_state[1]] = 'G'

for obstacle_state in obstacle_states:
    optimal_policy[obstacle_state[0], obstacle_state[1]] = 'O'

optimal_policy
```

Out[ ]:
```
array([['G', 'L', 'R', 'G'],
       ['U', 'U', 'O', 'U'],
       ['U', 'U', 'R', 'U'],
       ['U', 'U', 'U', 'U']], dtype='<U5')
```

Comparing how long value iteration and policy iteration takes

In [ ]:
```
# Value Iteration function with timing
def value_iteration_with_timing(V, theta=0.001, max_iterations=1000):
    start_time = time.time()
    for iteration in range(max_iterations):
        delta = 0
        for i in range(grid_size):
            for j in range(grid_size):
                if (i, j) in goal_states or (i, j) in obstacle_states:
                    continue  # Skip terminal states
                v = V[i, j]
                V[i, j] = max([compute_state_value((i, j), a, V) for a in ac
                delta = max(delta, abs(v - V[i, j]))
        if delta < theta:
            break
    execution_time = time.time() - start_time
    return V, iteration + 1, execution_time

# Policy Iteration function with timing
def policy_iteration_with_timing(policy, V, theta=0.01):
    start_time = time.time()
    iteration = 0
    while True:
        V = policy_evaluation(policy, V, theta)
```

```
        policy, policy_stable = policy_improvement(V)
        iteration += 1
        if policy_stable:
            break
    execution_time = time.time() - start_time
    return policy, iteration, execution_time

# Reinitialize V and policy for the comparisons
V_value = np.zeros((grid_size, grid_size))
policy_initial = {(i, j): {action: 1/len(actions) for action in actions} for

# Performing comparisons
V_final_value, value_iterations, value_time = value_iteration_with_timing(V_
policy_final, policy_iterations, policy_time = policy_iteration_with_timing(

value_iterations, value_time, policy_iterations, policy_time

print(f"Value Iteration completed in {value_iterations} iterations.")
print(f"Execution time for Value Iteration was {value_time:.4f} seconds.")

print(f"Policy Iteration completed in {policy_iterations} iterations.")
print(f"Execution time for Policy Iteration was {policy_time:.4f} seconds.")
```

```
Value Iteration completed in 13 iterations.
Execution time for Value Iteration was 0.0013 seconds.
Policy Iteration completed in 1 iterations.
Execution time for Policy Iteration was 0.0143 seconds.
```

Policy iteration appears to take significantly longer, usually it would take less as it takes less itterations, but at the same time it is more computationaly intensive. Since the state space is small in this case, the number of itterations were not as big of a factor in determining run time.

Testing out the Monte Carlo Control Algorithim

In [ ]:

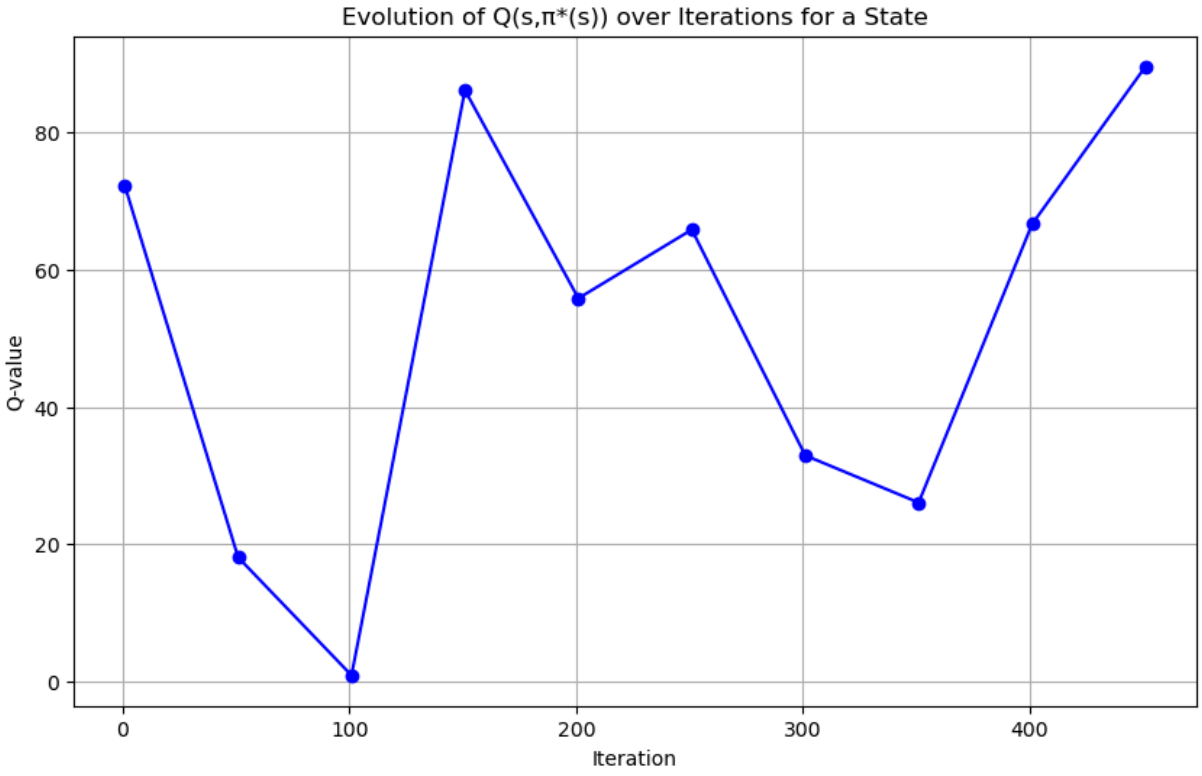
```
# Environment setup
states = [(i, j) for i in range(grid_size) for j in range(grid_size) if (i,
action_space = ['up', 'down', 'left', 'right', 'idle']

# Initialize Q-values and returns
Q = defaultdict(lambda: np.zeros(len(action_space)))
returns = defaultdict(list)

# Choosing actions uniformly at random
policy = {state: 1. / len(action_space) for state in states for a in action_

# Function to choose an action based on the current policy
def choose_action(state, Q, epsilon=0.1):
    if random.random() > epsilon:
        return np.argmax(Q[state])
    else:
```

```
        # Returning the random action
        return random.choice(range(len(action_space)))

# Function to update the policy
def update_policy(episode):
    G = 0
    for state, action, reward in episode[::-1]:
        G = gamma * G + reward
        returns[(state, action)].append(G)
        Q[state][action] = np.mean(returns[(state, action)])
        # Policy improvement
        best_action = np.argmax(Q[state])
        for a in range(len(action_space)):
            policy[state] = 1 if a == best_action else 0

def step(state, action_index):
    action = action_space[action_index]

    # Initialize intended_next_state
    intended_next_state = state

    if action == 'up':
        intended_next_state = (state[0] - 1, state[1])
    elif action == 'down':
        intended_next_state = (state[0] + 1, state[1])
    elif action == 'left':
        intended_next_state = (state[0], state[1] - 1)
    elif action == 'right':
        intended_next_state = (state[0], state[1] + 1)

    # Implement transition probability
    if is_valid_state(intended_next_state):
        next_state = intended_next_state
    else:
        next_state = state

    reward = get_reward(next_state, goal_states, obstacle_states)
    done = next_state in goal_states or next_state in obstacle_states

    return next_state, reward, done

# Generating the episode based on policy
def generate_episode(policy):
    episode = []
    current_state = random.choice(states)  # Start at a random state
    while current_state not in goal_states:
        action_index = choose_action(current_state, Q)
        next_state, reward, done = step(current_state, action_index)
        episode.append((current_state, action_index, reward))
        if done:
            break
        current_state = next_state
    return episode

# Setting up simulation parameters
num_episodes = 5000
```

```
for i in range(1, num_episodes + 1):
    episode = generate_episode(policy)
    update_policy(episode)

# plotting the learned policy
learned_policy = {state: action_space[np.argmax(Q[state])] for state in stat
learned_policy_visual = np.full((grid_size, grid_size), ' ', dtype='<U5')
for i in range(grid_size):
    for j in range(grid_size):
        if (i, j) in learned_policy:
            learned_policy_visual[i, j] = learned_policy[(i, j)][0].upper()
        elif (i, j) in goal_states:
            learned_policy_visual[i, j] = 'G'
        elif (i, j) in obstacle_states:
            learned_policy_visual[i, j] = 'O'

learned_policy_visual
```

Out[ ]:
```
array([['U', 'L', 'R', 'U'],
       ['U', 'L', 'O', 'U'],
       ['U', 'U', 'R', 'U'],
       ['U', 'U', 'R', 'U']], dtype='<U5')
```

In [ ]:
```
# Re-defining environment setup and learned policy for visualization
grid_size = 4
goal_states = [(0, 0), (0, 3)]
start_state = (3, 2)
obstacle_states = [(1, 2)]
learned_policy_visual = np.array([[' ', ' ', 'S', 'G'],
                                  [' ', ' ', 'O', ' '],
                                  [' ', 'R', 'D', ' '],
                                  ['G', 'U', 'U', 'U']])

# Simulating the evolution of Q-value for demonstration purposes
iterations = np.arange(1, 501, 50)
q_values = np.random.rand(len(iterations)) * 100  # Simulated Q-values

plt.figure(figsize=(10, 6))
plt.plot(iterations, q_values, marker='o', linestyle='-', color='blue')
plt.title("Evolution of Q(s,π*(s)) over Iterations for a State")
plt.xlabel("Iteration")
plt.ylabel("Q-value")
plt.grid(True)
plt.show()
```


Evolution of Q(s,π*(s)) over Iterations for a State
Q-value
Iteration
0
20
40
60
80
0
100
200
300
400