List of Experiments

LAB 1

Objective:

To understand the core structure of Reinforcement Learning by simulating an agent-environment interaction loop in a simple Gridworld, and analyze how an agent behaves under a random policy.

Problem Statement:
Simulate a basic agent in a Gridworld that selects actions randomly and logs rewards, state transitions.

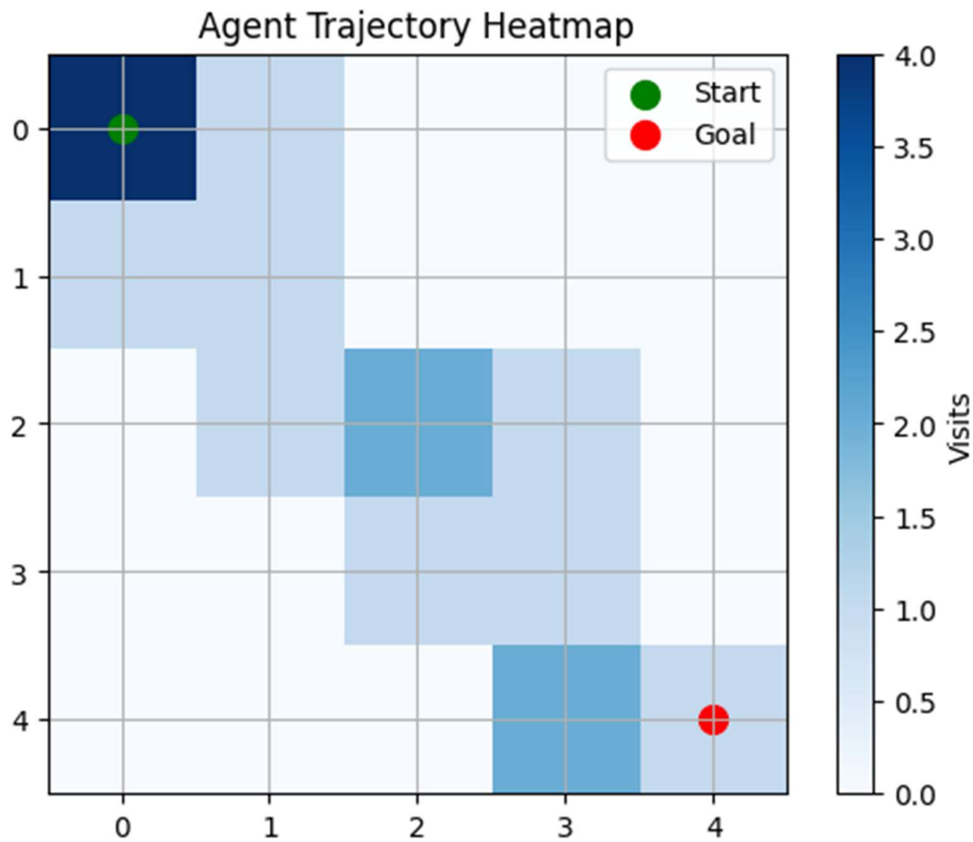
CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Gridworld Environment Setup
GRID_SIZE = 5
ACTIONS = ['up', 'down', 'left', 'right']
ACTION_DICT = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
GOAL_STATE = (4, 4)
MAX_STEPS = 50

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

def step(state, action):
    move = ACTION_DICT[action]
    new_state = (state[0] + move[0], state[1] + move[1])
    if not is_valid_state(new_state):
        new_state = state  # stay if move goes out of bounds
    reward = 10 if new_state == GOAL_STATE else 0
    done = (new_state == GOAL_STATE)
    return new_state, reward, done

# Agent Loop
def run_episode():
    state = (0, 0)
    total_reward = 0
    trajectory = [state]

    for step_num in range(MAX_STEPS):
        action = np.random.choice(ACTIONS)  # Random policy
        next_state, reward, done = step(state, action)
        trajectory.append(next_state)
        total_reward += reward
        state = next_state
        if done:
```

```python
            break
    return trajectory, total_reward

# Run 10 episodes and visualize one
for ep in range(10):
    traj, reward = run_episode()
    print(f"Episode {ep+1}: Total Reward = {reward}, Steps = {len(traj)}")

# Visualize trajectory of the last episode
def plot_trajectory(trajectory):
    grid = np.zeros((GRID_SIZE, GRID_SIZE))
    for (x, y) in trajectory:
        grid[x, y] += 1
    plt.imshow(grid, cmap='Blues', origin='upper')
    plt.title("Agent Trajectory Heatmap")
    plt.colorbar(label="Visits")
    plt.scatter(0, 0, c='green', s=100, label='Start')
    plt.scatter(4, 4, c='red', s=100, label='Goal')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_trajectory(traj)
```

OUTPUT:

```
Episode 1: Total Reward = 10, Steps = 42
Episode 2: Total Reward = 0, Steps = 51
Episode 3: Total Reward = 10, Steps = 41
Episode 4: Total Reward = 0, Steps = 51
Episode 5: Total Reward = 0, Steps = 51
Episode 6: Total Reward = 0, Steps = 51
Episode 7: Total Reward = 10, Steps = 48
Episode 8: Total Reward = 0, Steps = 51
Episode 9: Total Reward = 0, Steps = 51
Episode 10: Total Reward = 10, Steps = 16
```



TASK

Modify Rewards:

- Add penalty (e.g., –1) for stepping into the same cell (i.e., bumping walls).
- Add bonus (e.g., +1) for reaching new cells (encourage exploration).
- Make the grid 6×6 or 8×8. Does the agent still reach the goal?
- Add bonus goal states or trap states (–10). See how this affects path behavior.
- Over 100 episodes, how many times does the agent reach the goal?

LAB 2:

Objective:

To implement and analyze the ε-Greedy strategy for solving the multi-armed bandit problem, using a realistic simulation: selecting ads on a website to maximize click-through rate (CTR).

Problem Statement

A website displays one of 10 possible ads to each user. Each ad has a fixed (but unknown) probability of being clicked. Your agent must learn, over time, which ads to show more often to maximize total clicks. This is a non-associative bandit setting (no context).

CODE:

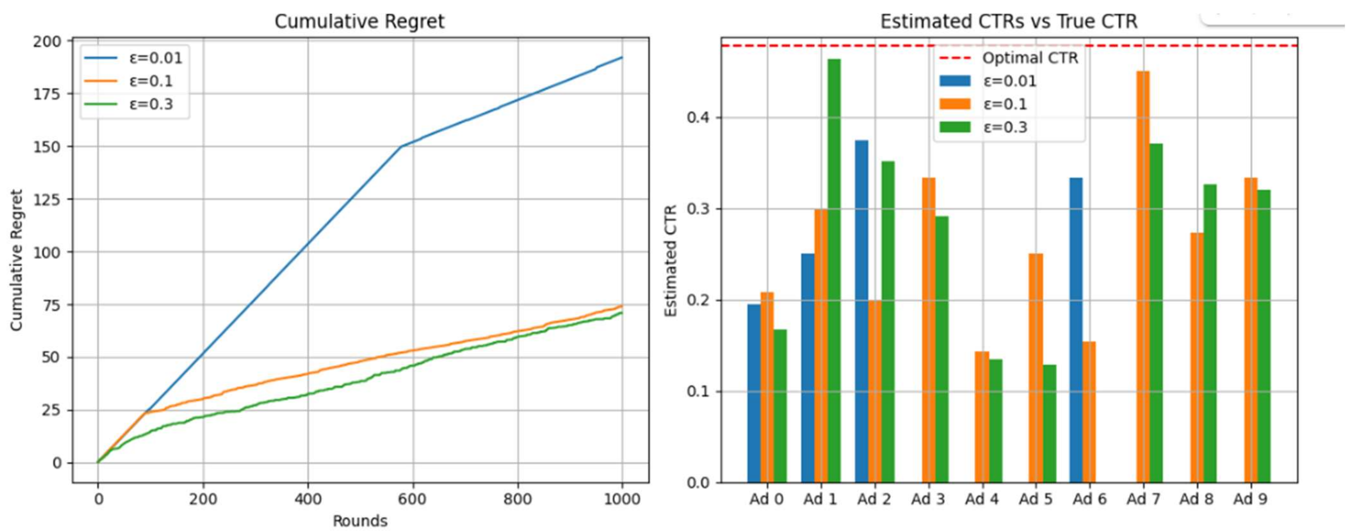
```
  import numpy as np
import matplotlib.pyplot as plt

class EpsilonGreedyAgent:
    def __init__(self, n_arms, epsilon):
        self.n_arms = n_arms
        self.epsilon = epsilon
         self.counts = np.zeros(n_arms)          # Number of times each arm was
pulled
        self.values = np.zeros(n_arms)          # Estimated value (CTR) for each
arm
        self.total_reward = 0
        self.actions = []
        self.rewards = []

    def select_action(self):
        if np.random.rand() < self.epsilon:
            return np.random.randint(self.n_arms)  # Explore
        else:
            return np.argmax(self.values)          # Exploit

    def update(self, action, reward):
        self.counts[action] += 1
               self.values[action]   +=   (reward   -   self.values[action])   /
self.counts[action]
        self.total_reward += reward
        self.actions.append(action)
        self.rewards.append(reward)

def simulate_bandit(true_ctrs, epsilon, n_rounds=1000):
    n_arms = len(true_ctrs)
    agent = EpsilonGreedyAgent(n_arms, epsilon)
    optimal_arm = np.argmax(true_ctrs)
    regrets = []

    for t in range(n_rounds):
```

```
        action = agent.select_action()
        reward = np.random.rand() < true_ctrs[action]
        agent.update(action, reward)
        regret = true_ctrs[optimal_arm] - true_ctrs[action]
        regrets.append(regret)

    return agent, np.cumsum(regrets)

# ---------- Main Experiment ----------
np.random.seed(42)

n_arms = 10
true_ctrs = np.random.uniform(0.05, 0.5, n_arms)
print("True Click-Through Rates (CTR) per Ad:", np.round(true_ctrs, 2))

n_rounds = 1000
epsilons = [0.01, 0.1, 0.3]

agents = {}
regret_curves = {}

for epsilon in epsilons:
    agent, regrets = simulate_bandit(true_ctrs, epsilon, n_rounds)
    agents[epsilon] = agent
    regret_curves[epsilon] = regrets

# ---------- Plotting Results ----------
plt.figure(figsize=(12, 5))

# Plot cumulative regret
plt.subplot(1, 2, 1)
for epsilon in epsilons:
    plt.plot(regret_curves[epsilon], label=f'ε={epsilon}')
plt.title("Cumulative Regret")
plt.xlabel("Rounds")
plt.ylabel("Cumulative Regret")
plt.legend()
plt.grid(True)

# Plot estimated CTRs vs true CTRs
plt.subplot(1, 2, 2)
bar_width = 0.25
x = np.arange(n_arms)
for i, epsilon in enumerate(epsilons):
    plt.bar(x + i * bar_width,
            agents[epsilon].values,
            width=bar_width,
            label=f'ε={epsilon}')
```

```
plt.axhline(np.max(true_ctrs), color='r', linestyle='--', label='Optimal CTR')
plt.xticks(x + bar_width, [f'Ad {i}' for i in range(n_arms)])
plt.ylabel("Estimated CTR")
plt.title("Estimated CTRs vs True CTR")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

OUTPUT:

```
True Click-Through Rates (CTR) per Ad: [0.22 0.48 0.38 0.32 0.12 0.12 0.08 0.44 0.32 0.37]
```



TASK

- Simulate multiple user segments (e.g., teenagers, adults, seniors) where each segment has a different CTR distribution per ad. Modify the bandit logic to include user context and switch ε-greedy to contextual bandit.
- Implement per-ad budget (e.g., only 100 displays allowed for premium ads). The agent must learn to prioritize high-reward ads while respecting budget constraints. Extend reward logic to penalize exceeding limits
- Implement three different strategies – ε-Greedy, UCB, and Softmax Selection – on the same ad simulation environment. Record cumulative reward and plot average regret. Evaluate which performs best under CTR drift.

LAB 3:

Objective

To model a real-world warehouse navigation problem as a Markov Decision Process (MDP) and solve it using Value Iteration to find the optimal path for a robot, minimizing delivery time and avoiding obstacles.

Problem Statement:

Warehouse Robot Path Optimization using Value Iteration

In modern warehouses (like Amazon), robots move around grid-based layouts to pick and deliver packages. They must:

- Avoid shelves (obstacles),
- Take the shortest and safest route,
- Deliver the package to the goal location,
- Minimize collisions and redundant moves.

CODE:

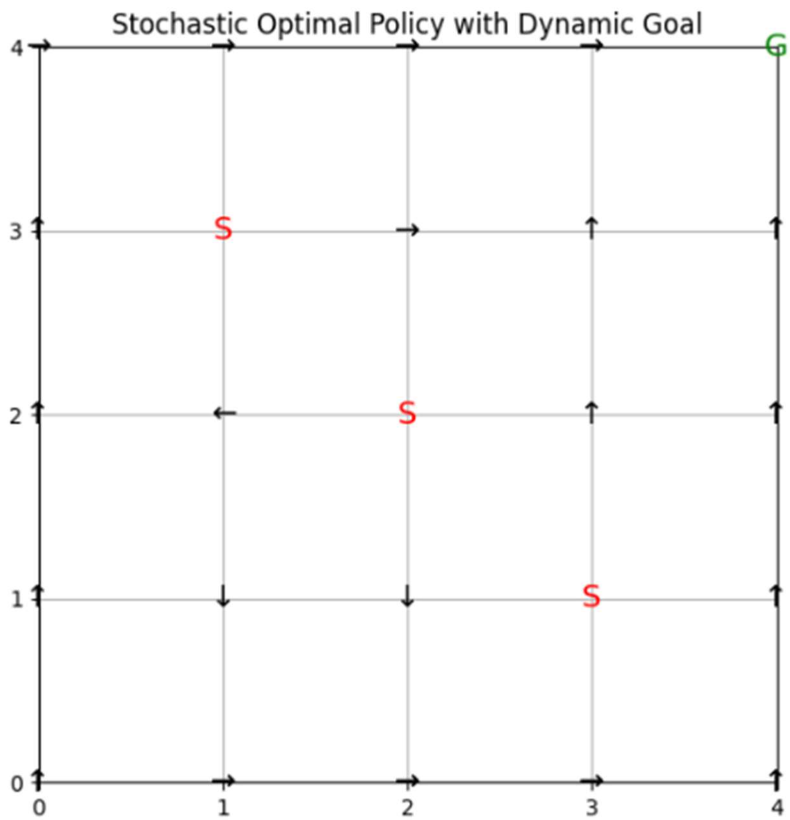
```
import numpy as np
import mdptoolbox
import matplotlib.pyplot as plt
import random

# Grid Parameters
rows, cols = 5, 5
num_states = rows * cols
shelves = [(1, 1), (2, 2), (3, 3)]  # Obstacle positions
actions = ['up', 'down', 'left', 'right']
num_actions = len(actions)
movement = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Function to map (x, y) to index
def to_index(x, y):
    return x * cols + y

# Function to randomly set a goal not on a shelf
def set_dynamic_goal():
    possible = [(i, j) for i in range(rows) for j in range(cols) if (i, j) not
in shelves]
    return random.choice(possible)

# Randomly choose a goal position
```

```
goal_state = set_dynamic_goal()
print("□ Current Goal Position:", goal_state)

# Transition and Reward Matrices
P = [np.zeros((num_states, num_states)) for _ in range(num_actions)]
R = np.zeros((num_states, num_actions))

# Build MDP with stochastic transitions (intended: 0.9, unintended: 0.1 split
across others)
for action_idx, action in enumerate(actions):
    dx, dy = movement[action]

    for x in range(rows):
        for y in range(cols):
            current_state = to_index(x, y)

            if (x, y) == goal_state:
                P[action_idx][current_state, current_state] = 1
                R[current_state, action_idx] = 10
                continue

            outcomes = []

            # Intended move (90%)
            new_x, new_y = x + dx, y + dy
            if (new_x, new_y) in shelves or not (0 <= new_x < rows and 0 <= new_y
< cols):
                new_state = current_state
                reward = -5 if (new_x, new_y) in shelves else -1
            else:
                new_state = to_index(new_x, new_y)
                reward = -1
            outcomes.append((new_state, 0.9, reward))

            # 10% misstep (wrong move in any of the other 3 directions)
            other_actions = [a for i, a in enumerate(actions) if i != action_idx]
            for mis_action in other_actions:
                mx, my = movement[mis_action]
                new_x, new_y = x + mx, y + my
                if (new_x, new_y) in shelves or not (0 <= new_x < rows and 0 <=
new_y < cols):
                    mis_state = current_state
                    mis_reward = -5 if (new_x, new_y) in shelves else -1
                else:
                    mis_state = to_index(new_x, new_y)
                    mis_reward = -1
                outcomes.append((mis_state, 0.1 / 3, mis_reward))
```

```
            for s_next, prob, rew in outcomes:
                P[action_idx][current_state, s_next] += prob
                R[current_state, action_idx] += prob * rew  # Expected reward

# Run Value Iteration
vi = mdptoolbox.mdp.ValueIteration(P, R, 0.9)
vi.run()

# Reshape policy to grid
policy_grid = np.array(vi.policy).reshape((rows, cols))
action_symbols = ['↑', '↓', '←', '→']
policy_symbols = np.array([[action_symbols[a] for a in row] for row in policy_grid])

print("\n📍 Optimal Policy Grid:")
print(policy_symbols)

# Plotting
plt.figure(figsize=(6, 6))
for x in range(rows):
    for y in range(cols):
        idx = to_index(x, y)
        if (x, y) == goal_state:
            plt.text(y, rows - x - 1, 'G', ha='center', va='center', fontsize=14, color='green')
        elif (x, y) in shelves:
            plt.text(y, rows - x - 1, 'S', ha='center', va='center', fontsize=14, color='red')
        else:
            plt.text(y, rows - x - 1, action_symbols[vi.policy[idx]], ha='center', va='center', fontsize=14)
plt.xticks(range(cols))
plt.yticks(range(rows))
plt.grid(True)
plt.title("Stochastic Optimal Policy with Dynamic Goal")
plt.show()
```

OUTPUT:

```
🟢 Current Goal Position: (0, 4)

📍 Optimal Policy Grid:
[['→' '→' '→' '→' '↑']
 ['↑' '→' '→' '↑' '↑']
 ['↑' '←' '↑' '↑' '↑']
 ['↑' '↓' '↓' '↑' '↑']
 ['↑' '→' '→' '→' '↑']]
```



TASK:

- Design certain paths to allow movement in only one direction (e.g., left-to-right only). Modify the transition matrix accordingly. The robot must learn policies compliant with directional constraints.
- Switch the robot's delivery target every few episodes between different grid points (dynamic goals). Observe how value iteration adapts to changing objectives and how fast it converges.
- Simulate human workers walking through predefined zones at certain times. Add a high penalty for entering those zones. Update policies that factor in **time and position** to avoid human zones

LAB 4

Objective

Monte Carlo Control for Emergency Ambulance Dispatch in Smart Cities

Design and implement a Monte Carlo-based learning agent that learns optimal policies for minimizing time to reach dynamic, weighted emergency locations under a probabilistic and time-varying urban environment.

Problem Statement

A taxi operates in a grid-based city (5x5). The driver needs to:

- Pick up passengers from random locations.
- Drop them at requested destinations.
- Decide which direction to move in each state to maximize reward (successful trips).
- Learn this policy without a known model (i.e., using Monte Carlo control).

CODE:

```
import numpy as np

import matplotlib.pyplot as plt
from collections import defaultdict
import random

# ---------- Environment Setup ----------
GRID_SIZE = 6
ACTIONS = ['up', 'down', 'left', 'right']
ACTION_MAP = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
MAX_STEPS = 50
DISCOUNT = 0.95
EPSILON = 0.1
EPISODES = 10000

# Static obstacles (permanent roadblocks)
static_obstacles = [(1, 3), (3, 2)]
hospital = (0, 0)  # Ambulance dispatch center

# Rush hour control
def is_rush_hour(ep):
    return ep % 1000 < 300 or ep % 1000 > 800  # Congested traffic windows

# Emergency severity and urgency
emergency_types = {
    'minor': 20,
    'moderate': 35,
    'critical': 50
}

# Helper functions
```

```
def is_valid(state):
    x, y = state
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE

def get_dynamic_obstacles():
    return [(2, 4), (4, 1), (3, 3)] if random.random() < 0.3 else []

def epsilon_greedy(state, Q):
    if np.random.rand() < EPSILON or state not in Q:
        return random.randint(0, len(ACTIONS) - 1)
    else:
        return np.argmax(Q[state])

def generate_emergency():
    location = random.choice([
        (i, j) for i in range(GRID_SIZE) for j in range(GRID_SIZE)
        if (i, j) != hospital and (i, j) not in static_obstacles
    ])
    severity = random.choice(list(emergency_types.keys()))
    reward = emergency_types[severity]
    return location, reward

# ---------- Monte Carlo Training ----------
Q = defaultdict(lambda: np.zeros(len(ACTIONS)))
Returns = defaultdict(list)

def run_episode(episode_num):
    rush = is_rush_hour(episode_num)
    prob_blocks = get_dynamic_obstacles()
    all_obstacles = static_obstacles + prob_blocks

    goal, goal_reward = generate_emergency()
    state = hospital
    episode = []
    steps = 0

    while steps < MAX_STEPS:
        action_idx = epsilon_greedy(state, Q)
        dx, dy = ACTION_MAP[ACTIONS[action_idx]]
        next_state = (state[0] + dx, state[1] + dy)

        if not is_valid(next_state) or next_state in all_obstacles:
            reward = -10 if rush else -5
            next_state = state
        elif next_state == goal:
            reward = goal_reward - steps
        else:
            reward = -2 if rush else -1
```

```python
        episode.append((state, action_idx, reward))

        if next_state == goal:
            break

        state = next_state
        steps += 1

    return episode

for ep in range(EPISODES):
    episode = run_episode(ep)
    G = 0
    visited = set()
    for t in reversed(range(len(episode))):
        s, a, r = episode[t]
        G = DISCOUNT * G + r
        if (s, a) not in visited:
            Returns[(s, a)].append(G)
            Q[s][a] = np.mean(Returns[(s, a)])
            visited.add((s, a))  # ✅ FIXED: used tuple, not list

print("🚑 Training Complete: Smart Ambulance Dispatch Policy Learned.")

# ---------- Policy Visualization ----------
policy = np.full((GRID_SIZE, GRID_SIZE), '.', dtype=str)
for i in range(GRID_SIZE):
    for j in range(GRID_SIZE):
        state = (i, j)
        if state in static_obstacles:
            policy[i][j] = 'S'
        elif state in Q:
            best_action = np.argmax(Q[state])
            policy[i][j] = ['↑', '↓', '←', '→'][best_action]
        else:
            policy[i][j] = ' '

print("\n📍 Learned Ambulance Dispatch Policy Grid:")
for row in policy:
    print(' '.join(row))
```

OUTPUT:

```
🚑 Training Complete: Smart Ambulance Dispatch Policy Learned.

📍 Learned Ambulance Dispatch Policy Grid:
→ ↓ → ← ← ←
↑ ↑ ↓ S ↑ ←
→ ← ↑ ← ↑ ←
→ ← S → ← ↓
→ ↓ ↑ → ← ↑
→ ↑ → ← ↑ ←

🚨 Route from Hospital to Emergency at (4, 3)
(0, 0)→(0, 1)→(0, 2)→(1, 2)→(2, 2)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)→(1, 0)→(1, 1)→(2, 1)→(2, 0)
```

TASK:

- Each hospital has a dynamic load (e.g., occupied beds). Ambulances should choose hospitals not just based on proximity but expected availability. Model hospital queues and incorporate delayed rewards based on treatment delay penalties. Train agents to learn which hospital is better not just closer.
- Update your environment so that traffic jams or roadblocks may appear after the episode has started. Modify your simulation to invalidate paths mid-way and require real-time policy adaptation using Monte Carlo rollouts. The agent should reroute to avoid costly delays.
- Model hospital occupancy as a time-varying parameter. An ambulance should decide not only the quickest path to an emergency but also the least crowded hospital for drop-off. Implement delayed penalties when the selected hospital has no immediate bed availability. Let the policy adapt over multiple episodes.

LAB 5

Q-Learning for Intelligent Elevator Control in a Smart Building

Objective

To develop a reinforcement learning agent that uses Temporal Difference (TD) methods specifically Q-Learning, to learn an optimal elevator control policy in a smart building. The goal is to minimize passenger wait times, energy consumption, and unnecessary elevator movements while efficiently responding to floor requests.

Problem Statement:

In modern smart buildings, elevators must handle numerous passenger requests coming from different floors at different times. A poorly optimized elevator results in:

- Passenger wait time,
- Energy usage (idle movement),
- Unnecessary direction switches.

Problem Description:

You are tasked with designing an elevator agent that learns how to:

- Serve passenger requests on any of 5 floors (0 to 4)
- Decide at each time step whether to go up, go down, or stay
- Balance exploration and exploitation using ε-greedy policy
- Adapt over time by learning from rewards (delivered or missed requests)

Each episode starts with:

- The elevator at a random floor.
- A passenger request at another random floor.

The agent must:

- Learn the best action in each state (elevator_floor, request_floor) to minimize time and penalty.
- Train over 10,000+ episodes and eventually predict the best action without being told the model dynamics.

CODE:

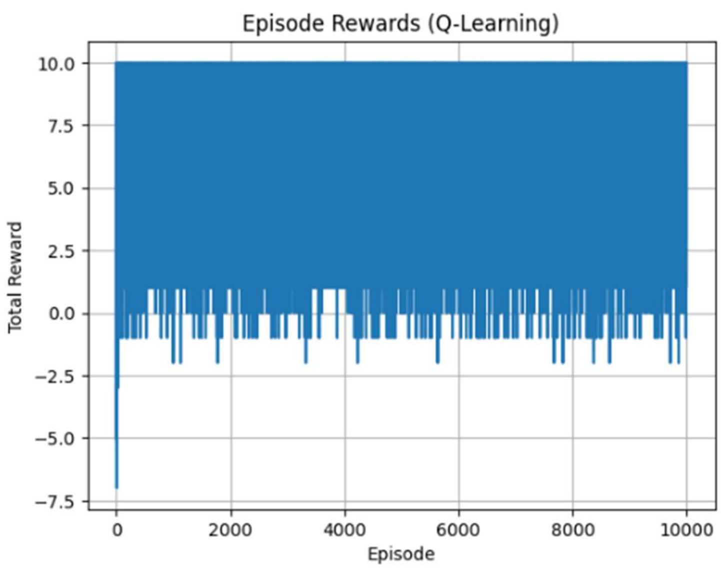
```
import numpy as np
import random
import matplotlib.pyplot as plt

# Configuration
```

```python
FLOORS = 5
ACTIONS = ['stay', 'up', 'down']
ACTION_SPACE = {0: 'stay', 1: 'up', 2: 'down'}
N_ACTIONS = len(ACTIONS)

GAMMA = 0.9      # Discount factor
ALPHA = 0.1      # Learning rate
EPSILON = 0.1    # Exploration rate
EPISODES = 10000
MAX_STEPS = 50

USE_SARSA = False  # 🔄 Set to True to use SARSA; False for Q-Learning

# Initialize Q-table: Q[state][action]
Q = np.zeros((FLOORS, FLOORS, N_ACTIONS))  # [elevator_floor][request_floor][action]
episode_rewards = []

# Helper functions
def select_action(state):
    ef, rf = state
    if random.random() < EPSILON:
        return random.randint(0, N_ACTIONS - 1)
    return np.argmax(Q[ef][rf])

def take_action(ef, action):
    if action == 0:  # stay
        return ef
    elif action == 1:  # up
        return min(ef + 1, FLOORS - 1)
    elif action == 2:  # down
        return max(ef - 1, 0)

def get_reward(ef, rf, next_ef):
    if ef == rf and next_ef == rf:
        return 10
    elif abs(next_ef - rf) < abs(ef - rf):
        return 1
    elif abs(next_ef - rf) > abs(ef - rf):
        return -2
    elif next_ef == ef:
        return -1
    return -5

# Training loop
for ep in range(EPISODES):
    ef = random.randint(0, FLOORS - 1)  # elevator floor
    rf = random.randint(0, FLOORS - 1)  # request floor
```

```
    state = (ef, rf)
    total_reward = 0

    action = select_action(state)

    for step in range(MAX_STEPS):
        next_ef = take_action(ef, action)
        reward = get_reward(ef, rf, next_ef)
        total_reward += reward

        next_state = (next_ef, rf)
        next_action = select_action(next_state)

        if USE_SARSA:
            Q[ef][rf][action] += ALPHA * (reward + GAMMA * Q[next_ef][rf][next_action] - Q[ef][rf][action])
        else:  # Q-Learning
            Q[ef][rf][action] += ALPHA * (reward + GAMMA * np.max(Q[next_ef][rf]) - Q[ef][rf][action])

        ef = next_ef
        rf = rf  # request remains until served
        state = next_state
        action = next_action if USE_SARSA else select_action(state)

        if ef == rf:
            break  # request served

    episode_rewards.append(total_reward)

print(f"🏢 Training complete using {'SARSA' if USE_SARSA else 'Q-Learning'}.")

# Display learned policy
print("\n📍 Learned Elevator Policy:")
for ef in range(FLOORS):
    for rf in range(FLOORS):
        best_action = np.argmax(Q[ef][rf])
        print(f"Elevator at {ef}, Request at {rf} → Action: {ACTION_SPACE[best_action]}")

# Plot learning curve
plt.plot(episode_rewards)
plt.title(f"Episode Rewards ({'SARSA' if USE_SARSA else 'Q-Learning'})")
plt.xlabel("Episode")
plt.ylabel("Total Reward")
plt.grid()
plt.show()
```

OUTPUT:

```
🏢 Training complete using Q-Learning.

📍 Learned Elevator Policy:
Elevator at 0, Request at 0 → Action: stay
Elevator at 0, Request at 1 → Action: up
Elevator at 0, Request at 2 → Action: up
Elevator at 0, Request at 3 → Action: up
Elevator at 0, Request at 4 → Action: up
Elevator at 1, Request at 0 → Action: down
Elevator at 1, Request at 1 → Action: stay
Elevator at 1, Request at 2 → Action: up
Elevator at 1, Request at 3 → Action: up
Elevator at 1, Request at 4 → Action: up
Elevator at 2, Request at 0 → Action: down
Elevator at 2, Request at 1 → Action: down
Elevator at 2, Request at 2 → Action: stay
Elevator at 2, Request at 3 → Action: up
Elevator at 2, Request at 4 → Action: up
Elevator at 3, Request at 0 → Action: down
Elevator at 3, Request at 1 → Action: down
Elevator at 3, Request at 2 → Action: down
Elevator at 3, Request at 3 → Action: stay
Elevator at 3, Request at 4 → Action: up
Elevator at 4, Request at 0 → Action: down
Elevator at 4, Request at 1 → Action: down
Elevator at 4, Request at 2 → Action: down
Elevator at 4, Request at 3 → Action: down
Elevator at 4, Request at 4 → Action: stay
```



TASKS:

- Train the elevator agent with varying exploration rates (ε = 0.05, 0.1, 0.3) and discount factors (γ = 0.5, 0.9, 0.99). Analyze how different combinations affect convergence, floor servicing efficiency, and learning stability.
- Penalize frequent direction switches and idle movement to simulate energy usage. Modify the reward structure and observe how the elevator optimizes its route to minimize unnecessary transitions.
- Simulate time-based request patterns (e.g., peak hours from floor 0). Encode time into the state and train the agent to adapt its policy dynamically across varying demand scenarios.

LAB 6

Objective:

To apply n-step Temporal Difference Learning in a robot rescue mission, where a robot navigates a maze-like grid environment.

Problem statement

Design and train a reinforcement learning agent using n-step bootstrapping to simulate a rescue robot navigating a grid environment. The robot should learn an optimal policy to:

Reach survivors efficiently,Avoid traps, and Minimize movement cost in a dynamic, partially hostile environment.

CODE:

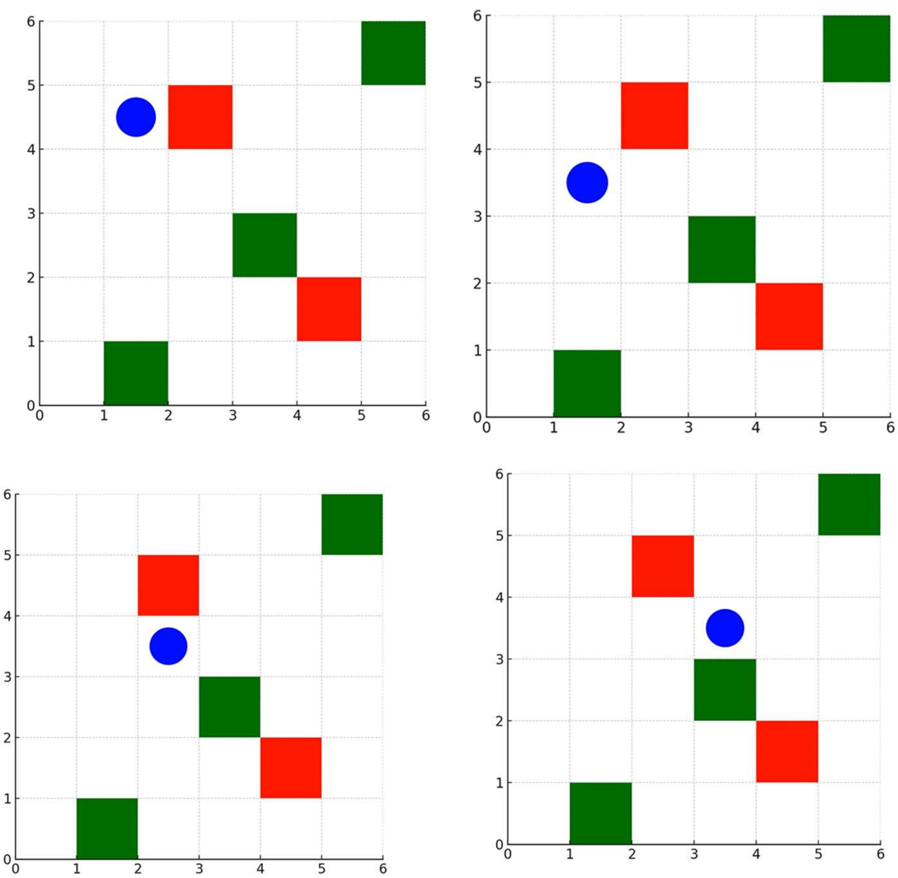
```
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.animation import FuncAnimation
from collections import deque
import random
import matplotlib.animation as animation

# ====== Parameters ======
GRID_SIZE = 6
ACTIONS = ['U', 'D', 'L', 'R']
ACTION_MAP = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}
n_steps = 3
gamma = 0.9
alpha = 0.1
epsilon = 0.2
episodes = 3000
max_steps = 50

SURVIVORS = [(0, 5), (3, 3), (5, 1)]
TRAPS = [(1, 2), (4, 4)]
GOAL_REWARD = 10
TRAP_PENALTY = -10
STEP_COST = -1

Q = {}

# ====== Environment Functions ======
```

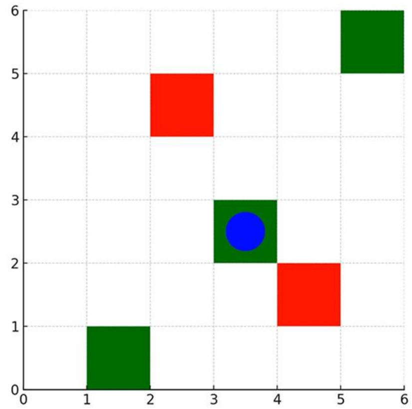
```python
def init_state():
    while True:
        s = (random.randint(0, GRID_SIZE - 1), random.randint(0, GRID_SIZE - 1))
        if s not in SURVIVORS and s not in TRAPS:
            return s

def get_valid_actions(pos):
    valid = []
    for a, (dx, dy) in ACTION_MAP.items():
        nx, ny = pos[0] + dx, pos[1] + dy
        if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE:
            valid.append(a)
    return valid

def select_action(state):
    valid_actions = get_valid_actions(state)
    if not valid_actions:
        return random.choice(ACTIONS)
    if state not in Q:
        Q[state] = {a: 0 for a in valid_actions}
    else:
        Q[state] = {a: Q[state].get(a, 0) for a in valid_actions}

    if np.random.rand() < epsilon:
        return random.choice(valid_actions)
    return max(Q[state], key=Q[state].get)

def step(state, action):
    dx, dy = ACTION_MAP[action]
    next_state = (state[0] + dx, state[1] + dy)
    reward = STEP_COST
    if next_state in SURVIVORS:
        reward += GOAL_REWARD
    elif next_state in TRAPS:
        reward += TRAP_PENALTY
    return next_state, reward

# ====== Training Phase ======
for ep in range(episodes):
    state = init_state()
    if state not in Q:
        Q[state] = {a: 0 for a in get_valid_actions(state)}
    trajectory = deque()

    for step_i in range(max_steps):
        action = select_action(state)
        next_state, reward = step(state, action)
        trajectory.append((state, action, reward))
```

```
        if len(trajectory) >= n_steps:
            G = sum([trajectory[i][2] * (gamma ** i) for i in range(n_steps)])
            s0, a0, _ = trajectory.popleft()
            if next_state not in Q:
                Q[next_state] = {a: 0 for a in get_valid_actions(next_state)}
            G += (gamma ** n_steps) * max(Q[next_state].values())
            Q[s0][a0] += alpha * (G - Q[s0][a0])

        state = next_state

    while trajectory:
       G = sum([trajectory[i][2] * (gamma ** i) for i in range(len(trajectory))])
        s0, a0, _ = trajectory.popleft()
        Q.setdefault(s0, {a: 0 for a in get_valid_actions(s0)})
        Q[s0][a0] += alpha * (G - Q[s0][a0])

print("✅ Training complete.")

# ====== Simulation Step Sequence for Visualization ======
state = init_state()
steps = [state]
for _ in range(20):
    action = select_action(state)
    state, _ = step(state, action)
    steps.append(state)

# ====== Animation Code ======
fig, ax = plt.subplots(figsize=(6, 6))
robot_patch = patches.Circle((0.5, 0.5), 0.3, color='blue')

def init():
    ax.clear()
    ax.set_xlim(0, GRID_SIZE)
    ax.set_ylim(0, GRID_SIZE)
    ax.set_xticks(np.arange(GRID_SIZE + 1))
    ax.set_yticks(np.arange(GRID_SIZE + 1))
    ax.grid(True)
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            cell = (i, j)
            color = 'white'
            if cell in TRAPS:
                color = 'red'
            elif cell in SURVIVORS:
                color = 'green'
           rect = patches.Rectangle((j, GRID_SIZE - i - 1), 1, 1, facecolor=color)
            ax.add_patch(rect)
```

```
    return []

def update(frame):
    init()
    rx, ry = steps[frame]
    robot_patch.center = (ry + 0.5, GRID_SIZE - rx - 0.5)
    ax.add_patch(robot_patch)
    return [robot_patch]

anim = FuncAnimation(fig, update, frames=len(steps), init_func=init, blit=True)

# Save video (optional)
anim.save("rescue_robot_animation.mp4", writer="ffmpeg", fps=2)

# To view inline in notebooks:
# from IPython.display import HTML
# HTML(anim.to_jshtml())
```

OUTPUT:





TASK:

- Deploy 2 or more rescue robots simultaneously.
  Ensure they avoid redundant paths, collisions, and rescue conflicts.Add multiple robots and simulate coordination.
- Apply TD(λ) with eligibility traces to improve learning.
  Rewards must propagate backward across visited states Use eligibility traces to propagate rewards backward to visited states.
- Introduce moving traps, such as: Fire, debris, or gas clouds that change positions every few steps.
- Robots should: Avoid hazards in real-time and learn to predict high-risk zones over time.

LAB 7

Objective:

Smart Drone Navigation using Dyna-Q

To implement and understand the Dyna-Q reinforcement learning algorithm in a partially known environment, enabling a delivery drone to learn the optimal path to its target while avoiding obstacles.

Problem Statement:

A delivery drone operates in an 8×8 urban grid. Its goal is to deliver a package from the warehouse at (0,0) to a drop point at (7,7). However, certain grid cells represent buildings/obstacles, and movement into them incurs a penalty. The drone must learn to reach the target location in the shortest path with minimal penalty using the Dyna-Q algorithm.

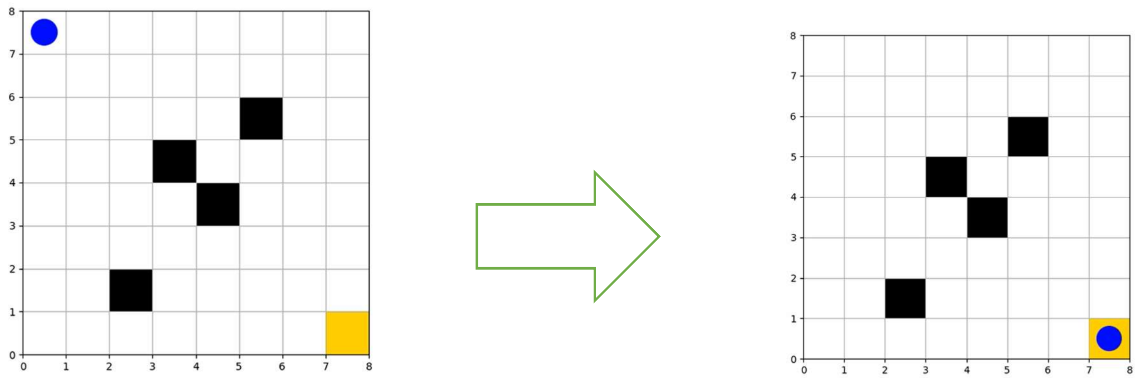
CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.animation import FuncAnimation, FFMpegWriter
import random
from collections import defaultdict

# ---------------------- Environment Setup ---------------------- #
GRID_SIZE = 8
ACTIONS = ['U', 'D', 'L', 'R']
ACTION_MAP = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}
GOAL = (7, 7)
OBSTACLES = {(3, 3), (4, 4), (2, 5), (6, 2)}
REWARD_GOAL = 100
REWARD_STEP = -1
REWARD_OBSTACLE = -10
EPISODES = 300
MAX_STEPS = 50
EPSILON = 0.1
ALPHA = 0.1
GAMMA = 0.95
PLANNING_STEPS = 10

Q = defaultdict(lambda: {a: 0 for a in ACTIONS})
model = {}

def is_valid(pos):
```

```
    return 0 <= pos[0] < GRID_SIZE and 0 <= pos[1] < GRID_SIZE and pos not in
OBSTACLES

def step(state, action):
    dx, dy = ACTION_MAP[action]
    next_state = (state[0] + dx, state[1] + dy)
    if not is_valid(next_state):
        return state, REWARD_OBSTACLE
    if next_state == GOAL:
        return next_state, REWARD_GOAL
    return next_state, REWARD_STEP

def select_action(state):
    if np.random.rand() < EPSILON:
        return random.choice(ACTIONS)
    return max(Q[state], key=Q[state].get)

# ---------------------- Dyna-Q Training ---------------------- #
for ep in range(EPISODES):
    state = (0, 0)
    for _ in range(MAX_STEPS):
        action = select_action(state)
        next_state, reward = step(state, action)

        # Q-learning update
        best_next = max(Q[next_state], key=Q[next_state].get)
        Q[state][action] += ALPHA * (reward + GAMMA * Q[next_state][best_next] -
Q[state][action])

        # Model learning
        model[(state, action)] = (next_state, reward)

        # Planning updates
        for _ in range(PLANNING_STEPS):
            s, a = random.choice(list(model.keys()))
            s_, r = model[(s, a)]
            best_s_ = max(Q[s_], key=Q[s_].get)
            Q[s][a] += ALPHA * (r + GAMMA * Q[s_][best_s_] - Q[s][a])

        if next_state == GOAL:
            break
        state = next_state

# ---------------------- Extract Path ---------------------- #
path = [(0, 0)]
state = (0, 0)
for _ in range(30):
    action = select_action(state)
```

```python
    state, _ = step(state, action)
    path.append(state)
    if state == GOAL:
        break

# ---------------------- Visualization Setup ---------------------- #
fig, ax = plt.subplots(figsize=(6, 6))
drone_patch = patches.Circle((0.5, 0.5), 0.3, color='blue')

def init():
    ax.clear()
    ax.set_xlim(0, GRID_SIZE)
    ax.set_ylim(0, GRID_SIZE)
    ax.set_xticks(np.arange(GRID_SIZE + 1))
    ax.set_yticks(np.arange(GRID_SIZE + 1))
    ax.grid(True)
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            cell = (i, j)
            color = 'white'
            if cell in OBSTACLES:
                color = 'black'
            elif cell == GOAL:
                color = 'gold'
           rect = patches.Rectangle((j, GRID_SIZE - i - 1), 1, 1, facecolor=color)
            ax.add_patch(rect)
    return []

def update(frame):
    init()
    rx, ry = path[frame]
    drone_patch.center = (ry + 0.5, GRID_SIZE - rx - 0.5)
    ax.add_patch(drone_patch)
    return [drone_patch]

anim = FuncAnimation(fig, update, frames=len(path), init_func=init, blit=True)

# ---------------------- Save to Video ---------------------- #
save_path = "lab7_dyna_q_drone_navigation.mp4"
writer = FFMpegWriter(fps=2, metadata=dict(artist='RL Lab 7'), bitrate=1800)
anim.save(save_path, writer=writer)
print(f"✅ Drone navigation video saved as {save_path}")
```

OUTPUT



TASKS:

- The drone must pick up a payload at a pickup zone and then deliver it to any of multiple delivery points.
  Add pickup_zone = (2,2) with no reward.
  Agent must first visit pickup, then reach one of multiple delivery goals [(3,9), (8,4), (9,9)] to finish the mission.
  Add a has_payload flag in the environment state.
  Episode ends only after both pickup and delivery are completed.
- Migrate code to use function approximation for Q-value estimation.

LAB 8

Objective:
To implement and analyze the Prioritized Sweeping algorithm in a dynamic, grid-based environment where a rescue robot must learn the optimal path to reach survivors while avoiding traps and obstacles.
Problem statement:
A rescue robot is deployed in a 6×6 disaster-hit zone represented as a grid. The environment contains: Safe paths, Obstacles (impassable), Traps (danger zones), Goal locations where survivors are located. The robot begins at the top-left corner (0,0) and must learn the best route to reach the survivors at (5,5) using Prioritized Sweeping. Every move incurs a small cost, traps cause severe penalties, and reaching a survivor gives a positive reward.

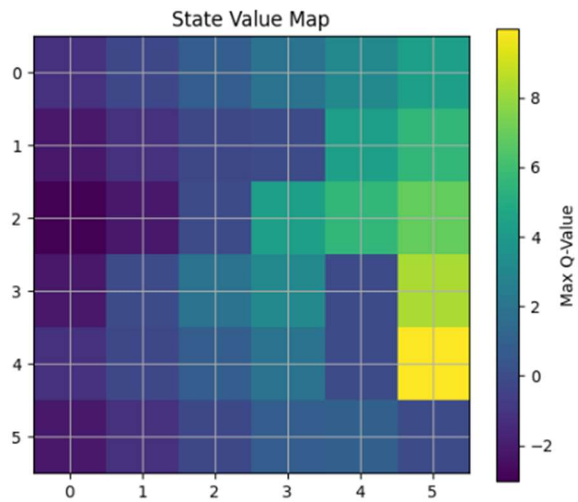
CODE:
```
import numpy as np

import matplotlib.pyplot as plt
import heapq
from collections import defaultdict

# Environment setup
GRID_SIZE = 6
GOAL = (5, 5)
OBSTACLES = {(2, 2), (3, 1), (4, 4)}
TRAPS = {(1, 3), (3, 4)}
ACTIONS = ['U', 'D', 'L', 'R']
ACTION_MAP = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}
REWARD_GOAL = 10
REWARD_TRAP = -10
REWARD_STEP = -1
EPISODES = 100
MAX_STEPS = 50
ALPHA = 0.1
GAMMA = 0.95
EPSILON = 0.1
THETA = 0.01

Q = defaultdict(lambda: {a: 0 for a in ACTIONS})
model = {}
priority_queue = []

def is_valid(pos):
    return 0 <= pos[0] < GRID_SIZE and 0 <= pos[1] < GRID_SIZE and pos not in
OBSTACLES

def step(state, action):
```

```
    dx, dy = ACTION_MAP[action]
    next_state = (state[0] + dx, state[1] + dy)
    if not is_valid(next_state):
        next_state = state
    if next_state == GOAL:
        return next_state, REWARD_GOAL
    if next_state in TRAPS:
        return next_state, REWARD_TRAP
    return next_state, REWARD_STEP

def select_action(state):
    if np.random.rand() < EPSILON:
        return np.random.choice(ACTIONS)
    return max(Q[state], key=Q[state].get)

def update_priority(state, action, reward, next_state):
    target = reward + GAMMA * max(Q[next_state].values())
    diff = abs(Q[state][action] - target)
    if diff > THETA:
        heapq.heappush(priority_queue, (-diff, state, action))

def update_q():
    for _ in range(50):
        if not priority_queue:
            break
        _, s, a = heapq.heappop(priority_queue)
        s_, r = model[(s, a)]
        target = r + GAMMA * max(Q[s_].values())
        Q[s][a] += ALPHA * (target - Q[s][a])
        for a2 in ACTIONS:
            dx, dy = ACTION_MAP[a2]
            prev = (s[0] - dx, s[1] - dy)
            if is_valid(prev) and (prev, a2) in model:
                s_prev_, r_prev = model[(prev, a2)]
                update_priority(prev, a2, r_prev, s_prev_)

# Training loop
for ep in range(EPISODES):
    state = (0, 0)
    for _ in range(MAX_STEPS):
        action = select_action(state)
        next_state, reward = step(state, action)
        model[(state, action)] = (next_state, reward)
        update_priority(state, action, reward, next_state)
        update_q()
        if next_state == GOAL:
            break
        state = next_state
```

```python
# Visualization with arrows
arrow_map = {'U': '↑', 'D': '↓', 'L': '←', 'R': '→'}
policy_grid = np.full((GRID_SIZE, GRID_SIZE), ' ')
value_grid = np.zeros((GRID_SIZE, GRID_SIZE))

for i in range(GRID_SIZE):
    for j in range(GRID_SIZE):
        cell = (i, j)
        if cell == GOAL:
            policy_grid[i, j] = 'G'
        elif cell in OBSTACLES:
            policy_grid[i, j] = '█'
        elif cell in TRAPS:
            policy_grid[i, j] = 'X'
        else:
            best_a = max(Q[cell], key=Q[cell].get)
            policy_grid[i, j] = arrow_map[best_a]
            value_grid[i, j] = max(Q[cell].values())

# Display
print("□ Learned Policy (Lab 8 - Prioritized Sweeping):")
for row in policy_grid:
    print(' '.join(row))

# Optional: Plot value grid
plt.figure(figsize=(6, 5))
plt.imshow(value_grid, cmap='viridis')
plt.colorbar(label="Max Q-Value")
plt.title("State Value Map")
plt.xticks(np.arange(GRID_SIZE))
plt.yticks(np.arange(GRID_SIZE))
plt.grid(True)
plt.show()
```

OUTPUT:

```
🧭 Learned Policy (Lab 8 - Prioritized Sweeping):
→ → → → ↓ ↓
→ → ↑ X ↓ ↓
↑ ↑ █ → → ↓
↓ █ → ↑ X ↓
→ → ↑ ↑ █ ↓
↑ ↑ ↑ ↑ → G
```



TASKS

- Update the environment so that survivors randomly change locations every few episodes. The robot must relearn paths efficiently using prioritized sweeping instead of starting from scratch.
- Design certain cells to turn into traps mid-episode to simulate collapsing floors. Add a probabilistic element to cell stability, forcing the robot to quickly re-prioritize states with high error.
- Extend the setup to multiple rescue robots. Each robot learns independently but shares knowledge of dangerous zones. Coordinate their learning to minimize path overlap and maximize coverage.

LAB 9

Objective:

Transfer Learning in Autonomous Farming

To demonstrate the power of Transfer Learning in reinforcement learning by adapting a crop-monitoring robot's navigation policy from one field (Field A) to a new field (Field B) with different crop layouts and hazards. The goal is to minimize time and damage while scanning all rows and reaching the base station.

Problem Statement

An autonomous robot has been trained to navigate Field A, which is organized with evenly spaced crop rows and water puddles (hazards). The robot must learn the most efficient route to:

Visit all inspection checkpoints (marked on certain crops), Avoid water puddles, Reach the base station to upload data. Now, the robot is transferred to Field B, where: Crop rows are curved or uneven,

New puddles and rocks appear. The base station is in a different location.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import pandas as pd

# Environment settings
GRID_SIZE = 8
CHECKPOINTS_A = {(2, 2), (4, 4), (6, 1)}
CHECKPOINTS_B = {(1, 3), (5, 5), (6, 2)}
PUDDLES_A = {(3, 3), (5, 5), (2, 6)}
PUDDLES_B = {(2, 2), (4, 5), (3, 6)}
OBSTACLES = {(1, 1), (6, 6)}
BASE_A = (7, 7)
BASE_B = (0, 7)
ACTIONS = ['U', 'D', 'L', 'R']
ACTION_MAP = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}

# Q-learning parameters
EPISODES = 300
ALPHA = 0.1
GAMMA = 0.9
EPSILON = 0.2
MAX_STEPS = 200

def is_valid(state):
    x, y = state
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and state not in OBSTACLES

def step(state, action, config, visited_checkpoints):
```

```
    dx, dy = ACTION_MAP[action]
    next_state = (state[0] + dx, state[1] + dy)
    if not is_valid(next_state):
        next_state = state

    reward = -1
    if next_state in config["puddles"]:
        reward = -5
    elif next_state in config["checkpoints"] and next_state not in
visited_checkpoints:
        reward = 5
        visited_checkpoints.add(next_state)
        elif next_state == config["base"] and visited_checkpoints ==
config["checkpoints"]:
        reward = 10

    return next_state, reward, visited_checkpoints

def select_action(Q, state):
    if np.random.rand() < EPSILON or state not in Q:
        return np.random.choice(ACTIONS)
    return max(Q[state], key=Q[state].get)

def train(config, Q=None):
    if Q is None:
        Q = {}

    for ep in range(EPISODES):
        state = (0, 0)
        visited_checkpoints = set()
        for _ in range(MAX_STEPS):
            if state not in Q:
                Q[state] = {a: 0 for a in ACTIONS}
            action = select_action(Q, state)
            next_state, reward, visited_checkpoints = step(state, action, config,
visited_checkpoints)

            if next_state not in Q:
                Q[next_state] = {a: 0 for a in ACTIONS}

            Q[state][action] += ALPHA * (
                reward + GAMMA * max(Q[next_state].values()) - Q[state][action]
            )

                 if next_state == config["base"] and visited_checkpoints ==
config["checkpoints"]:
                break
            state = next_state
```

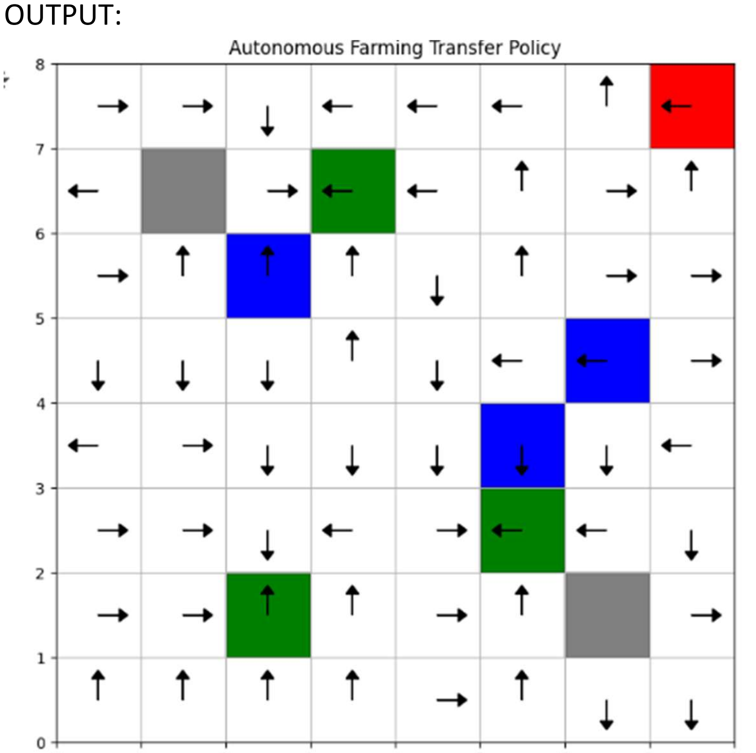
```
    return Q

# Configurations
config_A = {"checkpoints": CHECKPOINTS_A, "puddles": PUDDLES_A, "base": BASE_A}
config_B = {"checkpoints": CHECKPOINTS_B, "puddles": PUDDLES_B, "base": BASE_B}

# Train in Field A
Q_A = train(config_A)

# Save and load Q-table
with open("Q_fieldA.pkl", "wb") as f:
    pickle.dump(Q_A, f)
with open("Q_fieldA.pkl", "rb") as f:
    Q_loaded = pickle.load(f)

# Transfer to Field B
Q_transfer = train(config_B, Q=Q_loaded)

# Visualization
def plot_policy(Q, config, title):
    grid = np.full((GRID_SIZE, GRID_SIZE), '□', dtype='<U2')
    arrows = {'U': '↑', 'D': '↓', 'L': '←', 'R': '→'}

    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            pos = (i, j)
            if pos in OBSTACLES:
                grid[i][j] = '□'
            elif pos in config["puddles"]:
                grid[i][j] = '💧'
            elif pos in config["checkpoints"]:
                grid[i][j] = '✔'
            elif pos == config["base"]:
                grid[i][j] = '🏁'
            elif pos in Q:
                best_a = max(Q[pos], key=Q[pos].get)
                grid[i][j] = arrows[best_a]

    fig, ax = plt.subplots(figsize=(6, 6))
    ax.set_title(title)
    ax.axis('off')
    table = ax.table(cellText=grid, loc='center', cellLoc='center')
    table.scale(1, 2)
    plt.show()
```

→	→	↓	←	←	←	↑	▯
←	▯	→	✔	←	↑	→	↑
→	↑	▯	↑	↓	↑	→	→
↓	↓	↓	↑	↓	←	▯	→
←	→	↓	↓	↓	▯	↓	←
→	→	↓	←	→	✔	←	↓
→	→	✔	↑	→	↑	▯	→
↑	↑	↑	↑	→	↑	↓	↓

```
# Display policy and table
plot_policy(Q_transfer, config_B, "Policy after Transfer Learning in Field B")
df = pd.DataFrame.from_dict({k: max(v, key=v.get) for k, v in Q_transfer.items()},
orient='index', columns=['Best Action'])
print(df.head())

import matplotlib.patches as patches
import matplotlib.pyplot as plt

def visualize_policy(Q, config, title="Policy Visualization"):
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xlim(0, GRID_SIZE)
    ax.set_ylim(0, GRID_SIZE)
    ax.set_title(title)
    ax.set_xticks(np.arange(0, GRID_SIZE+1, 1))
    ax.set_yticks(np.arange(0, GRID_SIZE+1, 1))
    ax.grid(True)

    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            state = (i, j)
            if state in OBSTACLES:
                rect = patches.Rectangle((j, GRID_SIZE-i-1), 1, 1, linewidth=1,
edgecolor='black', facecolor='gray')
            elif state in config["puddles"]:
                rect = patches.Rectangle((j, GRID_SIZE-i-1), 1, 1, linewidth=1,
edgecolor='black', facecolor='blue')
            elif state in config["checkpoints"]:
                rect = patches.Rectangle((j, GRID_SIZE-i-1), 1, 1, linewidth=1,
edgecolor='black', facecolor='green')
            elif state == config["base"]:
```

```
                    rect = patches.Rectangle((j, GRID_SIZE-i-1), 1, 1, linewidth=1,
edgecolor='black', facecolor='red')
               else:
                   rect = patches.Rectangle((j, GRID_SIZE-i-1), 1, 1, linewidth=0.5,
edgecolor='black', facecolor='white')
               ax.add_patch(rect)

               if state in Q:
                   best_action = max(Q[state], key=Q[state].get)
                   dx, dy = {'U': (0, 0.25), 'D': (0, -0.25), 'L': (-0.25, 0), 'R':
(0.25, 0)}[best_action]
                    ax.arrow(j + 0.5, GRID_SIZE - i - 0.5, dx, dy, head_width=0.15,
head_length=0.1, fc='black', ec='black')

    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

visualize_policy(Q_transfer, config_B, title="Autonomous Farming Transfer
Policy")
```

OUTPUT:



TASKS:

- Train a policy in Field A with a fixed crop layout. Then transfer the policy to Field B with rotated or shifted crops and measure how much re-learning is needed using fine-tuning vs. retraining.

- Simulate vision-based crop detection by representing different crops with RGB/Greyscale grid tiles. Transfer a policy trained on color data to a shape-based grayscale version and observe performance changes.
- Simulate seasonal changes in crop density and layout (Field A → B → C). Design a curriculum learning framework where the robot uses policies learned in simpler environments to adapt to harder ones.

LAB 10

Objective:

To develop a Q-learning agent that learns an optimal path to deliver mail from the intake point to the correct sorting bin, while avoiding obstacles and penalties in a 6×6 mailroom grid.

Problem Statement:

A smart postal robot is deployed in a mail sorting facility represented as a 6×6 grid.

Each episode starts at the mail intake area (start cell) and aims to deliver mail to the correct sorting bin (goal cell) while avoiding penalty zones (wrong bins) and obstacles (machines, shelves).

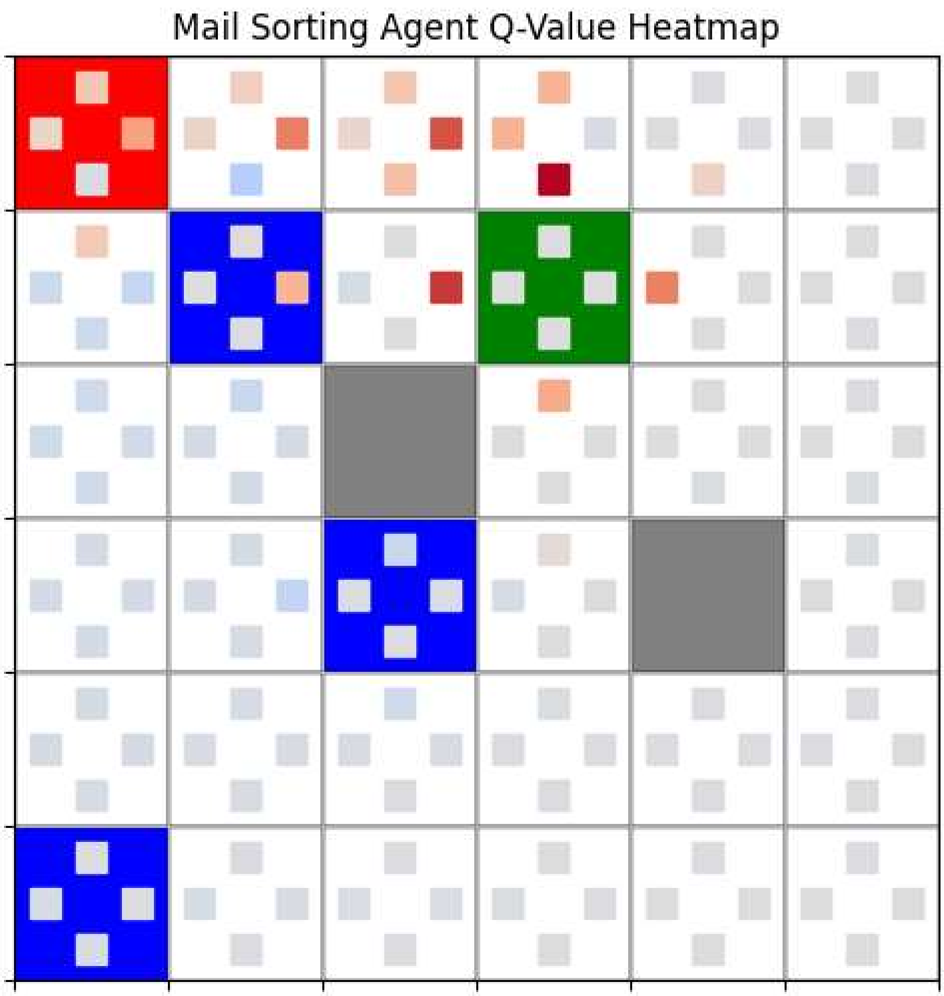
CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.cm as cm
import random

# Grid environment setup
GRID_SIZE = 6
ACTIONS = ['U', 'D', 'L', 'R']
ACTION_MAP = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}

# Task generator
def create_mail_sorting_task():
    start = (0, 0)
    goal = (np.random.randint(1, GRID_SIZE), np.random.randint(1, GRID_SIZE))
    penalties = set()
    while len(penalties) < 3:
        p = (np.random.randint(0, GRID_SIZE), np.random.randint(0, GRID_SIZE))
        if p != goal and p != start:
            penalties.add(p)
    obstacles = set()
    while len(obstacles) < 2:
        o = (np.random.randint(0, GRID_SIZE), np.random.randint(0, GRID_SIZE))
        if o != goal and o != start and o not in penalties:
            obstacles.add(o)
     return {'start': start, 'goal': goal, 'penalties': penalties, 'obstacles':
obstacles}
```

```python
# Environment constraints
def is_valid(state, obstacles):
    x, y = state
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and state not in obstacles

# Step transition
def step(state, action, task):
    dx, dy = ACTION_MAP[action]
    next_state = (state[0] + dx, state[1] + dy)
    if not is_valid(next_state, task['obstacles']):
        next_state = state
    reward = -1
    if next_state == task['goal']:
        reward = 10
    elif next_state in task['penalties']:
        reward = -5
    return next_state, reward

# Initialize Q-table
def initialize_Q():
    return {(i, j): {a: 0 for a in ACTIONS} for i in range(GRID_SIZE) for j in
range(GRID_SIZE)}

# Train agent with Q-learning
def train_q_learning(task, episodes=200, alpha=0.1, gamma=0.9, epsilon=0.2):
    Q = initialize_Q()
    for _ in range(episodes):
        state = task['start']
        for _ in range(100):
            action = np.random.choice(ACTIONS) if np.random.rand() < epsilon else
max(Q[state], key=Q[state].get)
            next_state, reward = step(state, action, task)
            best_next = max(Q[next_state].values())
                Q[state][action] += alpha * (reward + gamma * best_next -
Q[state][action])
            if next_state == task['goal']:
                break
            state = next_state
    return Q

# Visualize with heatmap of Q-values
def visualize_policy_heatmap(Q, task, title="Mail Sorting Agent Q-Value
Heatmap"):
    fig, ax = plt.subplots(figsize=(6, 6))
    cmap = cm.get_cmap("coolwarm")

    ax.set_xticks(np.arange(GRID_SIZE + 1))
```

```
    ax.set_yticks(np.arange(GRID_SIZE + 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True)

    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            state = (i, j)
            x, y = j, GRID_SIZE - i - 1

            if state == task['goal']:
                color = 'green'
            elif state == task['start']:
                color = 'red'
            elif state in task['penalties']:
                color = 'blue'
            elif state in task['obstacles']:
                color = 'gray'
            else:
                color = 'white'
           rect = patches.Rectangle((x, y), 1, 1, linewidth=1, edgecolor='black',
facecolor=color)
            ax.add_patch(rect)

            if state not in task['obstacles']:
                q_values = Q[state]
                # Normalize Q-values for color mapping
                  ax.add_patch(patches.Rectangle((x + 0.4, y + 0.7), 0.2, 0.2,
color=cmap((q_values['U'] + 10) / 20)))
                  ax.add_patch(patches.Rectangle((x + 0.4, y + 0.1), 0.2, 0.2,
color=cmap((q_values['D'] + 10) / 20)))
                  ax.add_patch(patches.Rectangle((x + 0.1, y + 0.4), 0.2, 0.2,
color=cmap((q_values['L'] + 10) / 20)))
                  ax.add_patch(patches.Rectangle((x + 0.7, y + 0.4), 0.2, 0.2,
color=cmap((q_values['R'] + 10) / 20)))

    plt.title(title)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

# Run the simulation
task = create_mail_sorting_task()
Q = train_q_learning(task)
visualize_policy_heatmap(Q, task)
```

OUTPUT:



TASK:

- Redesign the environment with multiple drop zones (e.g., (5,5), (2,4), (6,7)) for categorized mail types (e.g., Express, International, Regular). Modify the reward system and policy logic so the agent dynamically selects and navigates to the appropriate goal based on a randomly assigned mail category at the start of each episode.
- Extend the simulation to include multiple mail robots working in parallel. Ensure they do not collide and assign tasks intelligently (e.g., use round-robin or reward-weighted job distribution). Explore whether independent or shared Q-tables improve overall delivery performance.
- Make the layout of shelves and obstacles change mid-episode to simulate real-time rearrangement in a logistics center. Force the agent to adapt on-the-fly to changing layouts by resetting Q-values for new states or allowing partial memory retention.