import numpy as np  
import matplotlib.pyplot as plt  
import matplotlib.patches as patches  
import random  
  
# Grid world settings  
BOARD\_ROWS = 5  
BOARD\_COLS = 5  
WIN\_STATE = (4, 4)  
JUMP\_FROM = (1, 3)  
JUMP\_TO = (3, 3)  
START = (1, 0)  
OBSTACLES = [(2, 2), (2, 3), (2, 4), (3, 2)]  
  
DETERMINISTIC = False  
  
# Rewards  
REWARD\_WIN = 10  
REWARD\_JUMP = 5  
REWARD\_DEFAULT = -1  
  
# Actions: 1-North, 2-South, 3-East, 4-West  
ACTIONS = {  
 "1": (-1, 0),  
 "2": (1, 0),  
 "3": (0, 1),  
 "4": (0, -1)  
}  
  
ACTION\_NAMES = {  
 "1": "North",  
 "2": "South",  
 "3": "East",  
 "4": "West"  
}  
  
DIRECTION\_SYMBOLS = {  
 "1": "↑",  
 "2": "↓",  
 "3": "→",  
 "4": "←"  
}  
  
class State:  
 def \_\_init\_\_(self, state=START):  
 self.board = np.zeros([BOARD\_ROWS, BOARD\_COLS])  
 for (i, j) in OBSTACLES:  
 self.board[i, j] = -1  
 self.state = state  
 self.isEnd = False  
 self.determine = DETERMINISTIC  
  
 def giveReward(self):  
 if self.state == WIN\_STATE:  
 return REWARD\_WIN  
 else:  
 return REWARD\_DEFAULT  
  
 def isEndFunc(self):  
 if self.state == WIN\_STATE:  
 self.isEnd = True  
  
 def \_chooseActionProb(self, action):  
 if action == "1":  
 return np.random.choice(["1", "4", "3"], p=[0.8, 0.1, 0.1])  
 if action == "2":  
 return np.random.choice(["2", "4", "3"], p=[0.8, 0.1, 0.1])  
 if action == "3":  
 return np.random.choice(["3", "1", "2"], p=[0.8, 0.1, 0.1])  
 if action == "4":  
 return np.random.choice(["4", "1", "2"], p=[0.8, 0.1, 0.1])  
  
 def nxtPosition(self, action):  
 if not self.determine:  
 action = self.\_chooseActionProb(action)  
 self.determine = True  
 return self.nxtPosition(action)  
 else:  
 dr, dc = ACTIONS[action]  
 r, c = self.state  
 next\_r, next\_c = r + dr, c + dc  
  
 if next\_r < 0 or next\_r >= BOARD\_ROWS or next\_c < 0 or next\_c >= BOARD\_COLS:  
 return self.state  
  
 if (next\_r, next\_c) in OBSTACLES:  
 return self.state  
  
 if (next\_r, next\_c) == JUMP\_FROM:  
 return JUMP\_TO  
  
 return (next\_r, next\_c)  
  
  
class Agent:  
 def \_\_init\_\_(self, seed=42):  
 np.random.seed(seed)  
 random.seed(seed)  
 self.states = []  
 self.actions = ["1", "2", "3", "4"]  
 self.State = State()  
 self.isEnd = self.State.isEnd  
 self.lr = 1.0  
 self.exp\_rate = 0.3  
 self.decay\_gamma = 0.9  
 self.cumulative\_rewards = []  
  
 self.Q\_values = {}  
 for i in range(BOARD\_ROWS):  
 for j in range(BOARD\_COLS):  
 if (i, j) not in OBSTACLES:  
 self.Q\_values[(i, j)] = {a: 0.0 for a in self.actions}  
  
  
 def chooseAction(self):  
 if np.random.uniform(0, 1) <= self.exp\_rate:  
 return np.random.choice(self.actions)  
 else:  
 mx\_nxt\_reward = float('-inf')  
 action = np.random.choice(self.actions)  
 for a in self.actions:  
 current\_position = self.State.state  
 nxt\_reward = self.Q\_values[current\_position][a]  
 if nxt\_reward >= mx\_nxt\_reward:  
 action = a  
 mx\_nxt\_reward = nxt\_reward  
 return action  
  
 def takeAction(self, action):  
 next\_pos = self.State.nxtPosition(action)  
 self.states.append((self.State.state, action))  
 reward = REWARD\_DEFAULT ## Always step cost of -1 first  
 if next\_pos == WIN\_STATE:  
 reward = REWARD\_WIN  
 elif next\_pos == JUMP\_TO:  
 reward = REWARD\_JUMP  
 self.State.state = next\_pos  
 self.State.isEndFunc()  
 return reward  
  
  
 def reset(self):  
 self.states = []  
 self.State = State()  
 self.isEnd = self.State.isEnd  
  
 def play(self, rounds):  
 last\_30\_rewards = []  
 for episode in range(rounds):  
 self.reset()  
 cumulative\_reward = 0  
 while not self.State.isEnd:  
 current\_state = self.State.state  
 action = self.chooseAction()  
 reward = self.takeAction(action)  
 cumulative\_reward += reward  
  
 next\_state = self.State.state  
 current\_q = self.Q\_values[current\_state][action]  
 max\_future\_q = max(self.Q\_values[next\_state].values()) if next\_state in self.Q\_values else 0.0  
  
 # Q-learning  
 new\_q = current\_q + self.lr \* (reward + self.decay\_gamma \* max\_future\_q - current\_q)  
 self.Q\_values[current\_state][action] = round(new\_q, 3)  
  
 self.cumulative\_rewards.append(cumulative\_reward)  
 last\_30\_rewards.append(cumulative\_reward)  
  
 if len(last\_30\_rewards) > 30:  
 last\_30\_rewards.pop(0)  
  
 if len(last\_30\_rewards) == 30 and np.mean(last\_30\_rewards) >= 10.0:  
 print(f"Stopping early at Episode {episode + 1}, Avg Reward={np.mean(last\_30\_rewards):.2f}")  
 break  
 else:  
 action = self.chooseAction()  
 reward = self.takeAction(action)  
 cumulative\_reward += reward # accumulate all rewards, step + jump/win  
 self.State.isEndFunc()  
 self.isEnd = self.State.isEnd  
  
 self.state\_values = {  
 (i, j): max(self.Q\_values[(i, j)].values())  
 for (i, j) in self.Q\_values  
 }  
  
 def showbestPolicy(self):  
 for i in range(0, BOARD\_ROWS):  
 print('----------------------------------')  
 out = '| '  
 for j in range(0, BOARD\_COLS):  
 if (i, j) in OBSTACLES:  
 out += "###".ljust(6) + ' | '  
 else:  
 best\_action = max(self.Q\_values[(i, j)], key=self.Q\_values[(i, j)].get)  
 out += ACTION\_NAMES[best\_action].ljust(6) + ' | '  
 print(out)  
 print('----------------------------------')  
  
 def showStateValues(self): #Prints the learned values of all positions in a grid form.  
  
 for i in range(0, BOARD\_ROWS):  
 print('----------------------------------')  
 out = '| '  
 for j in range(0, BOARD\_COLS):  
 if (i, j) in OBSTACLES:  
 out += "###".ljust(6) + ' | '  
 else:  
 out += str(self.state\_values[(i, j)]).ljust(6) + ' | '  
 print(out)  
 print('----------------------------------')  
  
 def showQValues(self): #Prints the learned values of all positions in a grid form.  
 for i in range(0, BOARD\_ROWS):  
 print('----------------------------------')  
 out = ' | '  
 for j in range(0, BOARD\_COLS):  
 if (i, j) in OBSTACLES:  
 out += "###".ljust(6) + ' | '  
 else:  
 out += str(self.Q\_values[(i, j)]).ljust(6) + ' | '  
 print(out)  
 print('----------------------------------')  
  
 def plot\_rewards(self):  
 plt.plot(self.cumulative\_rewards)  
 plt.xlabel('Episode')  
 plt.ylabel('Cumulative Reward')  
 plt.title('Training Progress')  
 plt.grid()  
 plt.show()  
  
 def plot\_state\_values(self):  
 values = np.zeros((BOARD\_ROWS, BOARD\_COLS))  
 for (i, j) in self.Q\_values:  
 values[i, j] = max(self.Q\_values[(i, j)].values())  
  
 fig, ax = plt.subplots(figsize=(6, 6))  
 ax.set\_xlim(0, BOARD\_COLS)  
 ax.set\_ylim(0, BOARD\_ROWS)  
 ax.set\_xticks(np.arange(0, BOARD\_COLS+1))  
 ax.set\_yticks(np.arange(0, BOARD\_ROWS+1))  
 ax.grid(True)  
  
 for i in range(BOARD\_ROWS):  
 for j in range(BOARD\_COLS):  
 if (i, j) in OBSTACLES:  
 ax.add\_patch(patches.Rectangle((j, i), 1, 1, color='black'))  
 elif (i, j) == WIN\_STATE:  
 ax.add\_patch(patches.Rectangle((j, i), 1, 1, color='cyan'))  
 ax.text(j+0.5, i+0.5, '+10', ha='center', va='center')  
 elif (i, j) == JUMP\_FROM:  
 ax.text(j+0.5, i+0.5, '+5', ha='center', va='center', color='blue')  
 else:  
 ax.text(j+0.5, i+0.5, f"{values[i,j]:.1f}", ha='center', va='center')  
  
 plt.gca().invert\_yaxis()  
 plt.title("State Value Visualization")  
 plt.show()  
  
 def plot\_policy(self):  
 fig, ax = plt.subplots(figsize=(6, 6))  
 ax.set\_xlim(0, BOARD\_COLS)  
 ax.set\_ylim(0, BOARD\_ROWS)  
 ax.set\_xticks(np.arange(0, BOARD\_COLS+1))  
 ax.set\_yticks(np.arange(0, BOARD\_ROWS+1))  
 ax.grid(True)  
  
 for i in range(BOARD\_ROWS):  
 for j in range(BOARD\_COLS):  
 if (i, j) in OBSTACLES:  
 ax.add\_patch(patches.Rectangle((j, i), 1, 1, color='black'))  
 elif (i, j) == WIN\_STATE:  
 ax.add\_patch(patches.Rectangle((j, i), 1, 1, color='cyan'))  
 ax.text(j+0.5, i+0.5, 'WIN', ha='center', va='center')  
 elif (i, j) == JUMP\_FROM:  
 ax.text(j+0.5, i+0.5, 'Jump', ha='center', va='center', color='blue')  
 elif (i, j) in self.Q\_values:  
 best\_action = max(self.Q\_values[(i, j)], key=self.Q\_values[(i, j)].get)  
 ax.text(j+0.5, i+0.5, DIRECTION\_SYMBOLS[best\_action], ha='center', va='center', fontsize=14)  
  
 plt.gca().invert\_yaxis()  
 plt.title("Policy Visualization")  
 plt.show()  
  
  
# Main Execution  
if \_\_name\_\_ == "\_\_main\_\_":  
 ag = Agent()  
 print("Initial Q-values:")  
 print(ag.Q\_values)  
  
 ag.play(100)  
 print("\nLearned Best Actions (Policy):")  
 ag.showbestPolicy()  
  
 print("\nState Values Visualization:")  
 ag.showStateValues()  
  
 print("\nQ-Values Visualization:")  
 ag.showQValues()  
  
 ag.plot\_rewards()  
 ag.plot\_policy()  
 ag.plot\_state\_values()