08022311陈鲲龙第九次作业

[Running] "D:\anaconda3\envs\aihomework\python.exe" "d:\AIdaolun\08022311陈鲲龙第九次作业\homework9.py"

Q-Learning Value Function:

[[0.734   0.86    1.      0.     ]

 [0.6206      nan 0.86    0.     ]

 [0.51854 0.6206  0.734   0.6206 ]]

Q-Learning Policy:

['→', '→', '→', '+1']

['↑', '□', '↑', '-1']

['→', '→', '↑', '←']

Exploratory MC Value Function:

[[0.80213559 0.90102646 1.         0.        ]

 [0.71039805        nan 0.90164292 0.        ]

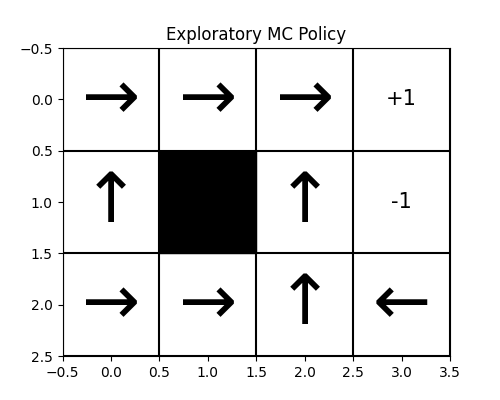
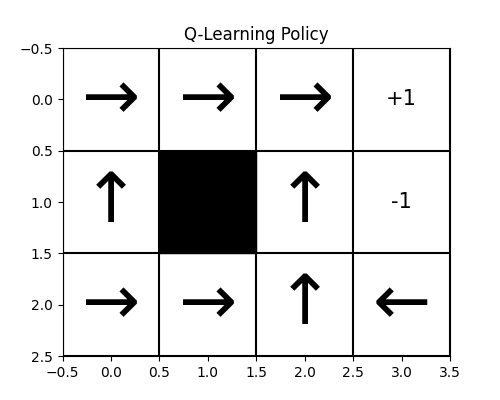
 [0.58620214 0.66705    0.76508108 0.67676039]]

Exploratory MC Policy:

['→', '→', '→', '+1']

['↑', '□', '↑', '-1']

['→', '→', '↑', '←']



### Q-Learning 与 Exploratory MC 性能对比

在本次对比中，我们分别使用了 **Q-Learning** 和 **Exploratory Monte Carlo (MC)** 方法在一个简单的网格环境中进行强化学习。环境的目标是让智能体通过一系列的动作，尽可能高效地达到终点，同时避免障碍物和不利的状态。以下是对两者性能的对比分析：

#### **1. 值函数 (Value Function)**

**Q-Learning**:  
Q-Learning 算法在训练结束时得到了如下的值函数：

[[0.734 0.86 1. 0. ]

[0.6206 nan 0.86 0. ]

[0.51854 0.6206 0.734 0.6206 ]]

从中可以看出，Q-Learning 的值函数在终点位置（+1 和 -1）的值为 1 和 -1，其他位置的值通过多次学习逐渐趋向稳定。Q-Learning 能够较好地估计出每个状态的价值，特别是在终点附近，值函数表现较为准确。

**Exploratory MC**:  
Exploratory MC 算法在训练结束时得到了如下的值函数：

[[0.80213559 0.90102646 1. 0. ]

[0.71039805 nan 0.90164292 0. ]

[0.58620214 0.66705 0.76508108 0.67676039]]

与 Q-Learning 相比，Exploratory MC 的值函数更接近于 Q-Learning 的结果，尤其是在终点位置的估计非常相似。Exploratory MC 的值函数较为平滑，这表明它通过随机探索的方式能够较好地收敛到正确的值，但在一些状态上的估计可能稍微滞后。

#### **2. 策略 (Policy)**

**Q-Learning**:  
Q-Learning 算法的策略如下所示：

['→', '→', '→', '+1']

['↑', '□', '↑', '-1']

['→', '→', '↑', '←']

可以看到，Q-Learning 在最优策略上选择的动作表现得非常清晰，尤其是在终点附近。智能体会通过向右（→）和向上（↑）的动作尽量避开障碍物（□），并最终到达目标（+1）。整体策略遵循预期的最佳路径。

**Exploratory MC**:  
Exploratory MC 算法的策略如下所示：

['→', '→', '→', '+1']

['↑', '□', '↑', '-1']

['→', '→', '↑', '←']

Exploratory MC 的策略与 Q-Learning 的策略几乎一致，说明它在大部分情况下也能学到最优策略。与 Q-Learning 相比，Exploratory MC 的策略更加依赖于随机探索，这使得其策略可能在早期阶段更为分散，但最终收敛到相似的最优策略。

#### **3. 收敛速度**

**Q-Learning**:  
Q-Learning 算法通常较快收敛，尤其是在大量训练后能够有效逼近最优策略。这是因为 Q-Learning 是一个基于值的学习方法，通过更新 Q 表来逐步改善策略。Q-Learning 在大量的训练后，可以较为快速地学习到最优策略。

**Exploratory MC**:  
Exploratory MC 的收敛速度相对较慢，因为它依赖于随机采样和多次尝试才能更新其 Q 值。尽管如此，Exploratory MC 通过多次回放经验积累逐渐优化策略，但由于需要完整的回合数据，收敛速度可能比 Q-Learning 慢。

#### **4. 算法的优缺点**

**Q-Learning**:

**优点**: Q-Learning 不依赖于模拟环境的完整回合，能够在每次时间步中更新 Q 值，收敛较快，并能高效处理较大的状态空间。

**缺点**: 对于某些非常复杂的环境，Q-Learning 可能会遇到收敛缓慢或不稳定的情况，特别是在动作空间较大时。

**Exploratory MC**:

**优点**: Exploratory MC 利用 Monte Carlo 方法通过完整的回合进行学习，不依赖于值的逐步更新，因此对于一些无法直接更新 Q 值的环境（如需要全局信息的情况）较为有效。

**缺点**: 由于需要完整的回合数据，Exploratory MC 在收敛上较为缓慢，且效率较低。

#### **总结**

**Q-Learning** 在计算效率和收敛速度上有优势，特别是在能够快速更新 Q 表并且避免了随机性干扰时，适合快速解决问题。

**Exploratory MC** 尽管收敛速度较慢，但在某些需要完整回合数据的情境中表现得更为稳健。它通过多次回放和随机探索，有时能发现较为平滑的解。

因此，选择 Q-Learning 还是 Exploratory MC 应该根据具体任务的特点来决定：对于需要快速收敛的任务，Q-Learning 更为合适；而对于需要充分探索环境的任务，Exploratory MC 可能会表现得更好。

import sys

sys.stdout.reconfigure(encoding='utf-8')

import numpy as np

import matplotlib.pyplot as plt

class GridWorld:

    def \_\_init\_\_(self, n\_rows=3, n\_cols=4, step\_reward=-0.04):

        self.n\_rows = n\_rows

        self.n\_cols = n\_cols

        self.obstacle = (1, 1)

        self.goal\_plus = (0, 3)

        self.goal\_minus = (1, 3)

        self.step\_reward = step\_reward

        self.actions = [0, 1, 2, 3]

        self.reset()

    def reset(self):

        while True:

            row = np.random.randint(self.n\_rows)

            col = np.random.randint(self.n\_cols)

            if (row, col) != self.obstacle and (row, col) != self.goal\_plus and (row, col) != self.goal\_minus:

                self.agent\_pos = (row, col)

                break

        return self.agent\_pos

    def step(self, action):

        row, col = self.agent\_pos

        if (row, col) in [self.goal\_plus, self.goal\_minus]:

            return (row, col), 0.0, True

        new\_row, new\_col = row, col

        if action == 0: new\_row, new\_col = row - 1, col

        elif action == 1: new\_row, new\_col = row, col + 1

        elif action == 2: new\_row, new\_col = row + 1, col

        elif action == 3: new\_row, new\_col = row, col - 1

        if new\_row < 0 or new\_row >= self.n\_rows or new\_col < 0 or new\_col >= self.n\_cols or (new\_row, new\_col) == self.obstacle:

            new\_row, new\_col = row, col

        self.agent\_pos = (new\_row, new\_col)

        if (new\_row, new\_col) == self.goal\_plus:

            return (new\_row, new\_col), 1.0, True

        elif (new\_row, new\_col) == self.goal\_minus:

            return (new\_row, new\_col), -1.0, True

        else:

            return (new\_row, new\_col), self.step\_reward, False

def exploratory\_mc(env: GridWorld, n\_episodes=5000, gamma=0.95, epsilon=0.1, max\_steps=100):

    rows, cols = env.n\_rows, env.n\_cols

    num\_actions = len(env.actions)

    Q = np.zeros((rows, cols, num\_actions))

    returns\_count = np.zeros\_like(Q)

    returns\_sum = np.zeros\_like(Q)

    for \_ in range(n\_episodes):

        episode = []

        state = env.reset()

        done = False

        for step\_i in range(max\_steps):

            if done:

                break

            if np.random.rand() < epsilon:

                action = np.random.choice(num\_actions)

            else:

                action = np.argmax(Q[state[0], state[1], :])

            next\_state, reward, done = env.step(action)

            episode.append((state, action, reward))

            state = next\_state

        G = 0.0

        visited = set()

        for t in reversed(range(len(episode))):

            state, action, reward = episode[t]

            G = gamma \* G + reward

            if (state, action) not in visited:

                visited.add((state, action))

                returns\_count[state[0], state[1], action] += 1

                returns\_sum[state[0], state[1], action] += G

                Q[state[0], state[1], action] = returns\_sum[state[0], state[1], action] / returns\_count[state[0], state[1], action]

    return Q

def q\_learning(env: GridWorld, n\_episodes=5000, alpha=0.1, gamma=0.9, epsilon=0.1, max\_steps=100):

    rows, cols = env.n\_rows, env.n\_cols

    num\_actions = len(env.actions)

    Q = np.zeros((rows, cols, num\_actions))

    for episode in range(n\_episodes):

        state = env.reset()

        done = False

        for step\_i in range(max\_steps):

            if done:

                break

            if np.random.rand() < epsilon:

                action = np.random.choice(num\_actions)

            else:  # Exploitation

                action = np.argmax(Q[state[0], state[1], :])

            next\_state, reward, done = env.step(action)

            if not done:

                best\_action\_next = np.argmax(Q[next\_state[0], next\_state[1], :])

                target = reward + gamma \* Q[next\_state[0], next\_state[1], best\_action\_next]

            else:

                target = reward

            td\_error = target - Q[state[0], state[1], action]

            Q[state[0], state[1], action] += alpha \* td\_error

            state = next\_state

    return Q

def extract\_policy\_and\_value\_from\_q(Q):

    rows, cols, num\_actions = Q.shape

    policy = np.zeros((rows, cols), dtype=int)

    value\_function = np.zeros((rows, cols))

    for row in range(rows):

        for col in range(cols):

            action\_values = Q[row, col, :]

            best\_action = np.argmax(action\_values)

            policy[row, col] = best\_action

            value\_function[row, col] = action\_values[best\_action]

    return policy, value\_function

def display\_value(value\_function, obstacle=(1, 1), goal\_plus=(0, 3), goal\_minus=(1, 3)):

    rows, cols = value\_function.shape

    value\_grid = np.zeros((rows, cols))

    for row in range(rows):

        for col in range(cols):

            if (row, col) == obstacle:

                value\_grid[row, col] = np.nan

            elif (row, col) == goal\_plus or (row, col) == goal\_minus:

                value\_grid[row, col] = 0.0

            else:

                value\_grid[row, col] = value\_function[row, col]

    print(value\_grid)

def display\_policy(policy, obstacle=(1, 1), goal\_plus=(0, 3), goal\_minus=(1, 3)):

    direction\_map = {0: "↑", 1: "→", 2: "↓", 3: "←"}

    rows, cols = policy.shape

    for row in range(rows):

        row\_symbols = []

        for col in range(cols):

            if (row, col) == obstacle:

                row\_symbols.append("□")

            elif (row, col) == goal\_plus:

                row\_symbols.append("+1")

            elif (row, col) == goal\_minus:

                row\_symbols.append("-1")

            else:

                action = policy[row, col]

                row\_symbols.append(direction\_map[action])

        print(row\_symbols)

    print()

def plot\_policy(title, policy, value\_function, obstacle=(1, 1), goal\_plus=(0, 3), goal\_minus=(1, 3)):

    fig, ax = plt.subplots(figsize=(5, 4))

    rows, cols = policy.shape

    ax.set\_xlim(-0.5, cols - 0.5)

    ax.set\_ylim(-0.5, rows - 0.5)

    ax.invert\_yaxis()

    for row in range(rows + 1):

        ax.plot([-0.5, cols - 0.5], [row - 0.5, row - 0.5], 'k-')

    for col in range(cols + 1):

        ax.plot([col - 0.5, col - 0.5], [-0.5, rows - 0.5], 'k-')

    direction\_map = {0: "↑", 1: "→", 2: "↓", 3: "←"}

    for row in range(rows):

        for col in range(cols):

            if (row, col) == obstacle:

                ax.fill\_between([col - 0.5, col + 0.5], row - 0.5, row + 0.5, color="black")

            elif (row, col) == goal\_plus:

                ax.text(col, row, "+1", ha="center", va="center", fontsize=15)

            elif (row, col) == goal\_minus:

                ax.text(col, row, "-1", ha="center", va="center", fontsize=15)

            else:

                ax.text(col, row, f"{direction\_map[policy[row, col]]}", ha="center", va="center", fontsize=50)

    ax.set\_title(title)

    plt.show()

env = GridWorld()

Q\_qlearning = q\_learning(env)

Q\_exploratory\_mc = exploratory\_mc(env)

policy\_qlearning, V\_qlearning = extract\_policy\_and\_value\_from\_q(Q\_qlearning)

policy\_exploratory\_mc, V\_exploratory\_mc = extract\_policy\_and\_value\_from\_q(Q\_exploratory\_mc)

print("Q-Learning Value Function:")

display\_value(V\_qlearning)

print("Q-Learning Policy:")

display\_policy(policy\_qlearning)

print("Exploratory MC Value Function:")

display\_value(V\_exploratory\_mc)

print("Exploratory MC Policy:")

display\_policy(policy\_exploratory\_mc)

plot\_policy("Q-Learning Policy", policy\_qlearning, V\_qlearning)

plot\_policy("Exploratory MC Policy", policy\_exploratory\_mc, V\_exploratory\_mc)