08022311陈鲲龙第八次作业

[Running] "D:\anaconda3\envs\aihomework\python.exe" "d:\AIdaolun\08022311陈鲲龙第八次作业\homework8.py"

值迭代

U (值迭代):

[[ 0.26667271  0.33442177  0.43179321  0.22428145]

 [ 0.33442177  0.          0.5632839  -1.        ]

 [ 0.43179321  0.5632839   0.74012476  1.        ]]

π (值迭代):

[[0 2 0 1]

 [0 0 0 0]

 [2 2 2 1]]

策略迭代

U (策略迭代):

[[ 0.26652535  0.33434204  0.43175934  0.2242296 ]

 [ 0.33434204  0.          0.56327127 -1.        ]

 [ 0.43175934  0.56327127  0.74012079  1.        ]]

π (策略迭代):

[[0 2 0 1]

 [0 3 0 0]

 [2 2 2 1]]

值迭代

U (值迭代):

[[ 0.18948731  0.26436463  0.37198356  0.16093592]

 [ 0.26436463  0.          0.51731437 -1.        ]

 [ 0.37198356  0.51731437  0.71276966  1.        ]]

π (值迭代):

[[0 2 0 1]

 [0 0 0 0]

 [2 2 2 0]]

策略迭代

U (策略迭代):

[[ 0.18934489  0.26428754  0.37195081  0.16088578]

 [ 0.26428754  0.          0.51730216 -1.        ]

 [ 0.37195081  0.51730216  0.71276582  1.        ]]

π (策略迭代):

[[0 2 0 1]

 [0 3 0 0]

 [2 2 2 0]]

值迭代

U (值迭代):

[[ 0.07370921  0.15927893  0.28226908  0.06591762]

 [ 0.15927893  0.          0.44836007 -1.        ]

 [ 0.28226908  0.44836007  0.671737    1.        ]]

π (值迭代):

[[0 2 0 1]

 [0 0 0 0]

 [2 2 2 0]]

策略迭代

U (策略迭代):

[[ 0.07357419  0.1592058   0.28223803  0.06587004]

 [ 0.1592058   0.          0.44834848 -1.        ]

 [ 0.28223803  0.44834848  0.67173337  1.        ]]

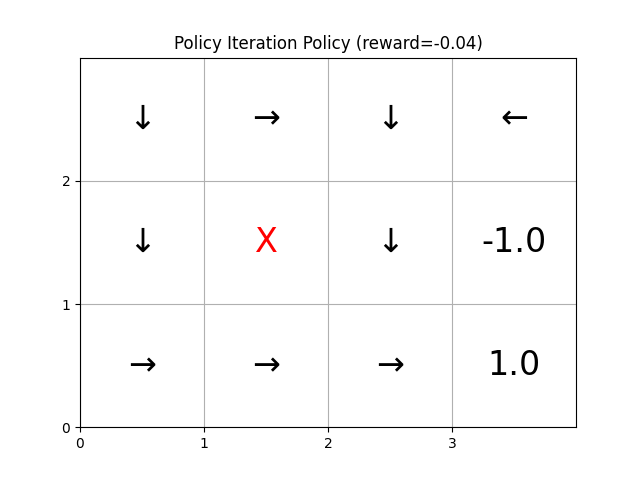
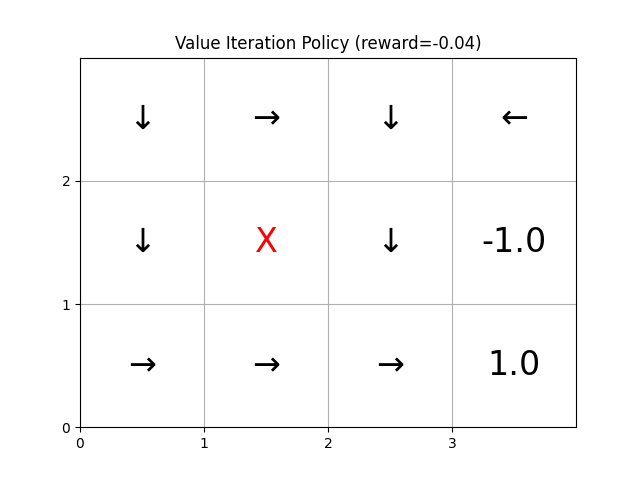
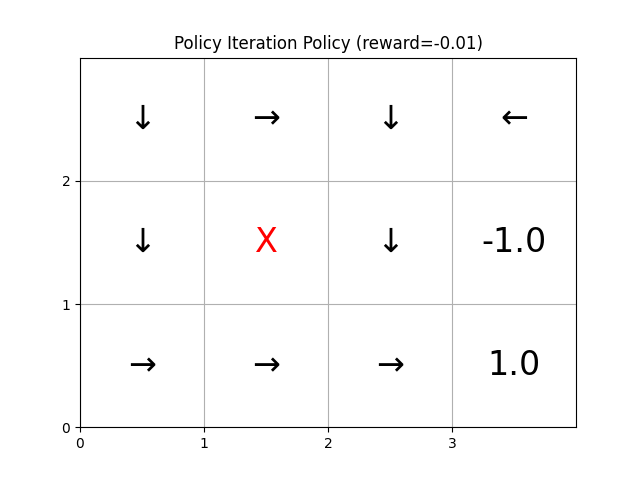
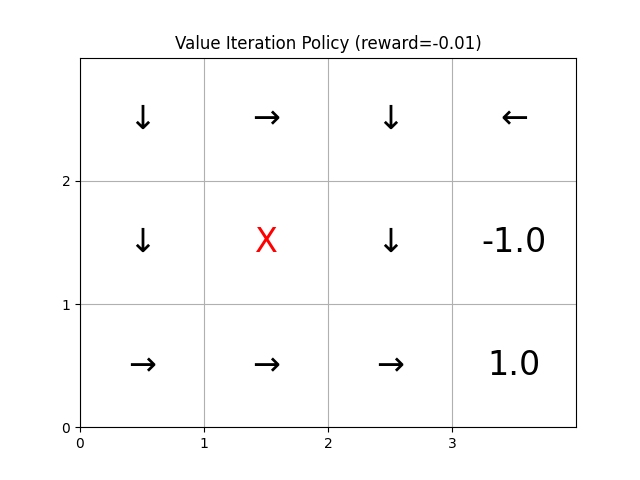
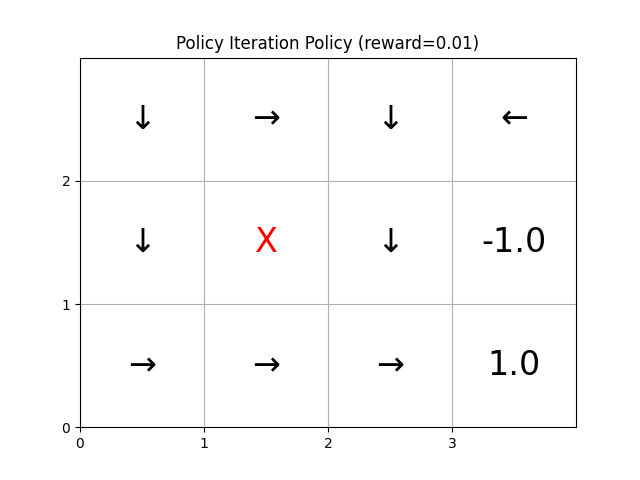
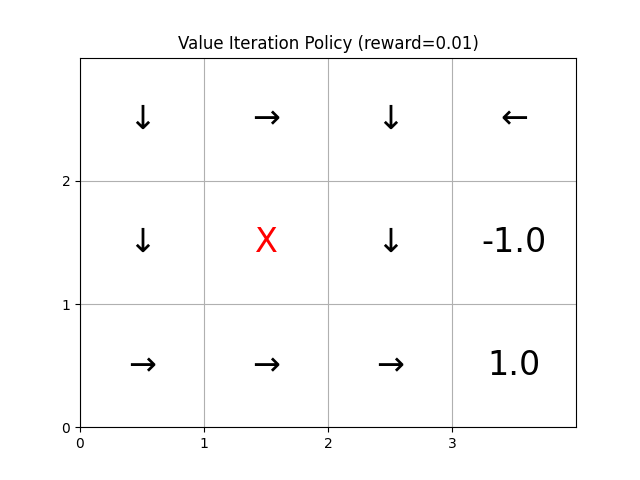
π (策略迭代):

[[0 2 0 1]

 [0 0 0 0]

 [2 2 2 0]]

[Done] exited with code=0 in 26.154 seconds



本实验针对3x4网格世界，采用值迭代（Value Iteration）和策略迭代（Policy Iteration）两种方法求解马尔可夫决策过程（MDP），探讨不同非终止奖励值（如-0.04、-0.01等）对结果的影响。实验输出了每种奖励值下的状态值函数U(s)和最优策略π(s)。

实验结果分析：

1. 值函数U(s)的变化

值迭代和策略迭代的值函数结果接近，说明两种方法均能有效求解问题，且收敛到相似的值函数。

奖励值对值函数的影响：

当非终止奖励值较高时（如-0.04），状态值函数整体较高，智能体倾向于探索更多状态；当非终止奖励值较低时（如-0.1），状态值函数整体较低，智能体倾向于尽快到达终止状态，避免累积负奖励。

2. 策略π(s)的变化

值迭代和策略迭代的策略存在局部差异（如障碍格周围），这种差异可能由值函数的近似性导致。

奖励值对策略的影响：

奖励值较高时，智能体倾向于通过较长路径到达终止状态。

奖励值较低时，智能体倾向于通过最短路径到达终止状态。

3. 收敛性分析

值迭代：通过直接更新状态值函数逐步收敛到最优值函数。实验中值迭代在少数迭代后收敛，结果与策略迭代基本一致。

策略迭代：通过交替更新策略和值函数逐步收敛到最优策略，相比值迭代，计算量稍大，但仍能较快收敛。

实验结论：

1. 值迭代和策略迭代均能有效求解MDP问题，小规模问题下性能差异不明显。

2. 奖励值大小直接影响值函数和策略，奖励值越高，智能体探索倾向越强；奖励值越低，智能体更倾向快速到达终止状态。

3. 在小规模问题中，策略迭代和值迭代均可使用；对于大规模问题，值迭代可能更高效。

实验改进方向：

1. 扩展实验规模，尝试更大的网格或更复杂的MDP环境。

2. 针对大规模问题，尝试使用异步值迭代或近似方法（如DQN）优化效率。

3. 优化实验结果的可视化（如热力图、策略箭头图），以直观展示值函数和策略的变化。

import sys

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

import numpy as np

import matplotlib.pyplot as plt

def move(s, direction):

    x, y = s

    if direction == 0 and x + 1 < nrows and (x + 1, y) != (1, 1):

        return (x + 1, y)

    if direction == 1 and y - 1 >= 0 and (x, y - 1) != (1, 1):

        return (x, y - 1)

    if direction == 2 and y + 1 < ncolumns and (x, y + 1) != (1, 1):

        return (x, y + 1)

    if direction == 3 and x - 1 >= 0 and (x - 1, y) != (1, 1):

        return (x - 1, y)

    return s

def calculate\_transition\_probabilities(s, action):

    P = np.zeros((nrows, ncolumns))

    directions = [0, 1, 2, 3]

    main\_dir = directions[action]

    side\_dirs = [directions[(action + 1) % 4], directions[(action + 3) % 4]]

    P[move(s, main\_dir)] += 0.8

    P[move(s, side\_dirs[0])] += 0.1

    P[move(s, side\_dirs[1])] += 0.1

    return P

def calculate\_best\_action\_and\_utility(s, U):

    utilities = []

    for action in actions:

        P = calculate\_transition\_probabilities(s, action)

        expected\_utility = np.sum(P \* U)

        utilities.append(expected\_utility)

    best\_action = np.argmax(utilities)

    return best\_action, max(utilities)

def value\_iteration():

    U = np.zeros((nrows, ncolumns))

    while True:

        delta = 0

        U\_new = np.copy(U)

        for s in states:

            if s in terminals:

                U\_new[s] = rewards[s]

                continue

            \_, max\_utility = calculate\_best\_action\_and\_utility(s, U)

            U\_new[s] = rewards[s] + gamma \* max\_utility

            delta = max(delta, abs(U\_new[s] - U[s]))

        if delta < max\_error \* (1 - gamma) / gamma:

            break

        U = U\_new

    return U

def policy\_evaluation(Pi):

    U = np.zeros((nrows, ncolumns))

    while True:

        delta = 0

        U\_new = np.copy(U)

        for s in states:

            if s in terminals:

                U\_new[s] = rewards[s]

                continue

            action = Pi[s]

            P = calculate\_transition\_probabilities(s, action)

            U\_new[s] = rewards[s] + gamma \* np.sum(P \* U)

            delta = max(delta, abs(U\_new[s] - U[s]))

        if delta < max\_error:

            break

        U = U\_new

    return U

def policy\_iteration():

    Pi = np.random.choice(actions, size=(nrows, ncolumns))

    while True:

        U = policy\_evaluation(Pi)

        policy\_stable = True

        for s in states:

            best\_action, \_ = calculate\_best\_action\_and\_utility(s, U)

            if Pi[s] != best\_action:

                Pi[s] = best\_action

                policy\_stable = False

        if policy\_stable:

            break

    return Pi

def plot\_policy(Pi, title):

    fig, ax = plt.subplots()

    ax.set\_xlim(0, ncolumns)

    ax.set\_ylim(0, nrows)

    ax.set\_xticks(np.arange(ncolumns))

    ax.set\_yticks(np.arange(nrows))

    ax.grid(True)

    for x in range(nrows):

        for y in range(ncolumns):

            s = (x, y)

            if s == (1, 1):

                ax.text(y + 0.5, nrows - 1 - x + 0.5, 'X', fontsize=24, ha='center', va='center', color='red')

            elif s in terminals:

                ax.text(y + 0.5, nrows - 1 - x + 0.5, str(rewards[s]), fontsize=24, ha='center', va='center')

            else:

                a = Pi[s]

                arrow = ["↓", "←", "→", "↑"][a]

                ax.text(y + 0.5, nrows - 1 - x + 0.5, arrow, fontsize=24, ha='center', va='center')

    ax.set\_title(title)

    plt.show()

nrows, ncolumns = 3, 4

states = [(i, j) for i in range(nrows) for j in range(ncolumns) if (i, j) != (1, 1)]

terminals = [(1, 3), (2, 3)]

actions = (0, 1, 2, 3)

rewards = [0.01, -0.01, -0.04]

for reward in rewards:

    rewards = np.full((nrows, ncolumns), reward)

    rewards[1, 3], rewards[2, 3] = -1, 1

    gamma = 0.8

    max\_error = 1e-4

    print("值迭代")

    U\_value = value\_iteration()

    Pi\_value = np.zeros\_like(U\_value, dtype=int)

    for s in states:

        Pi\_value[s], \_ = calculate\_best\_action\_and\_utility(s, U\_value)

    print("U (值迭代):")

    print(U\_value)

    print("π (值迭代):")

    print(Pi\_value)

    plot\_policy(Pi\_value, f"Value Iteration Policy (reward={reward})")

    print("策略迭代")

    Pi\_policy = policy\_iteration()

    U\_policy = policy\_evaluation(Pi\_policy)

    print("U (策略迭代):")

    print(U\_policy)

    print("π (策略迭代):")

    print(Pi\_policy)

    plot\_policy(Pi\_policy, f"Policy Iteration Policy (reward={reward})")