强化学习基础算法实验报告

姓名：________ 张韫译萱________ 学号：________08023214________

一、 实验题目

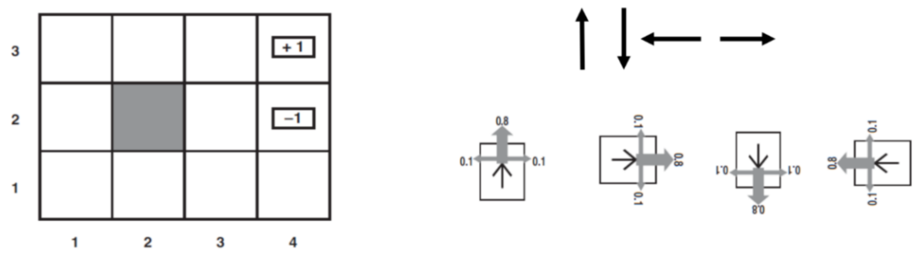

图1 状态转移概率模型

图 1: 状态转移模型

1 实验任务

1. 实现值迭代与策略迭代（模型 P 已知），对三种（非终止状态）奖赏值 $R(S) = \{0.01, -0.01, -0.04\}$，找出最优策略。
2. 实现 Exploratory-MC 与 Q-Learning（模型 P 未知），非终止状态的奖赏值为 $R(s) = -0.04$，并实验对比它们的性能。

2 实验要求

1. 选择 C++ 或 Python 实现。
2. 代码以文本形式粘贴在附录相应位置，注释准确，能成功运行。

二、 实验结果

1 实验参数设置

本实验中自定义的参数如下：折扣因子 $\gamma = 0.9$，该值能够平衡即时奖励与长期收益；收敛阈值 $\theta = 10^{-4}$，作为值迭代和策略评估的终止条件；Q-Learning 学习率 $\alpha = 0.1$；探索率 $\epsilon = 0.1$；MC 和 Q-Learning 均训练 10000 轮。

2 值迭代与策略迭代运行结果

对三种不同的每步奖赏值 $R(s)$ 进行实验，得到的最优策略如下。箭头表示该状态下的最优动作，背景颜色深浅代表状态价值。

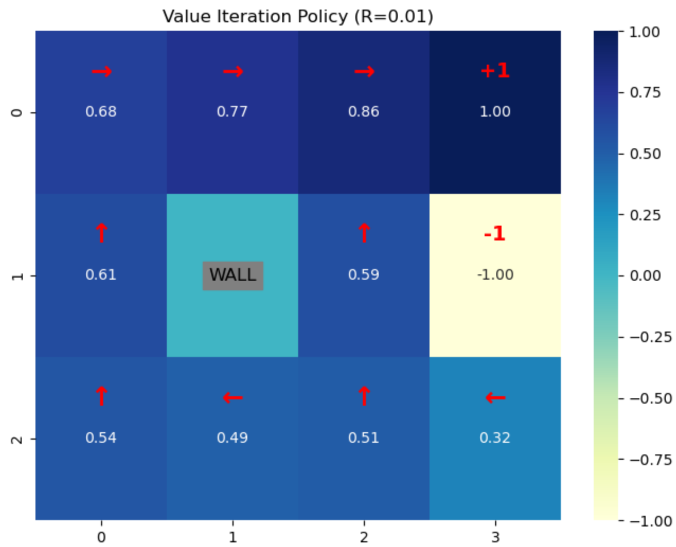

图 2: R(s) = 0.01 时的最优策略与价值函数

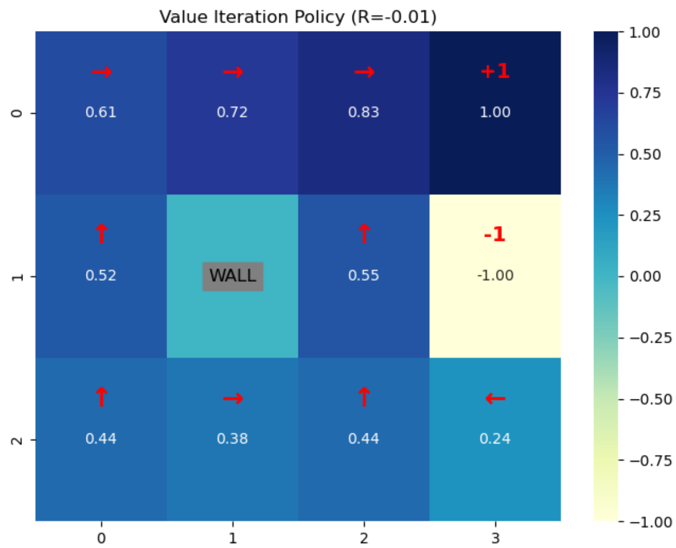

图 3: R(s) = -0.01 时的最优策略与价值函数

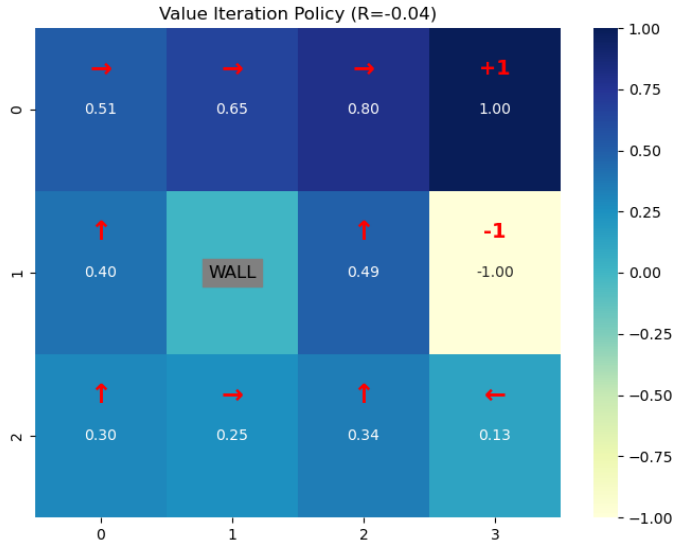

图 4: R(s) = -0.04 时的最优策略与价值函数

3 Exploratory-MC 与 Q-Learning 运行结果

在 $R(s) = -0.04$ 的情况下，分别使用蒙特卡洛（Monte Carlo ES）和 Q-Learning 算法进行学习。

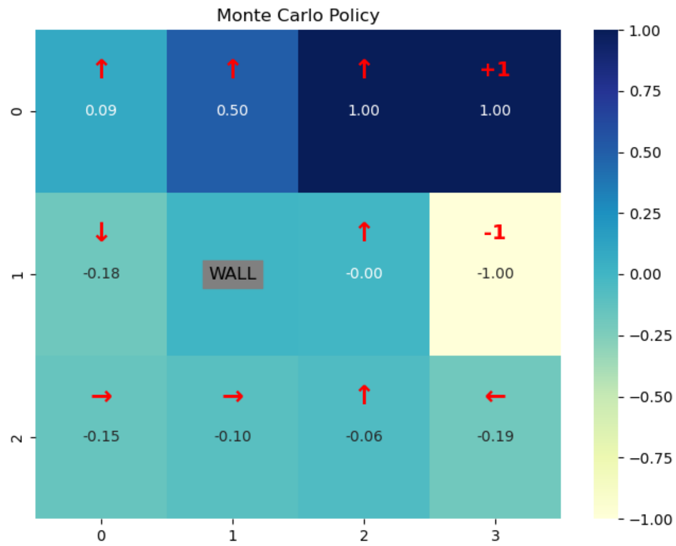

图 5: Monte Carlo ES 学习到的策略

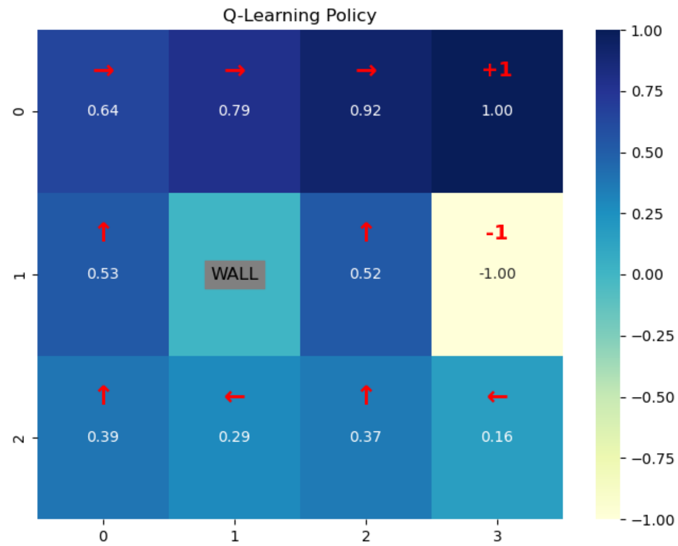

图 6: Q-Learning 学习到的策略

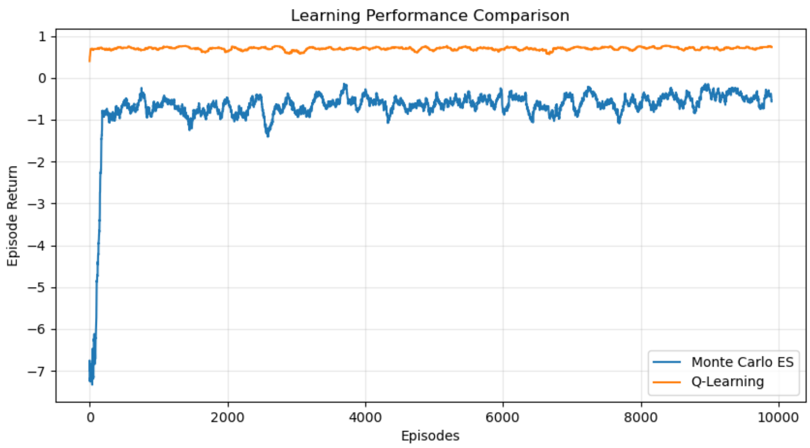

图 7: Monte Carlo ES 与 Q-Learning 学习曲线对比

三、 实验分析

1 值迭代与策略迭代分析

实验结果表明，步奖励 $R(s)$ 的设置对最优策略具有显著影响。

当 $R(s) = 0.01$ 时，非终止状态具有正奖励，智能体倾向于延长轨迹长度以累积更多奖励。策略图中可观察到部分状态的动作指向远离终止状态的方向，这是因为在折扣因子 $\gamma = 0.9$ 的作用下，持续获取正奖励的累积收益可能超过直接到达 $+1$ 终止状态的收益。

当 $R(s) = -0.01$ 时，每步存在轻微惩罚，智能体开始倾向于寻找通往 $+1$ 终止状态的路径。但

由于惩罚较小，智能体会优先选择风险较低的路径，避免靠近-1 终止状态。例如，(2,1) 位置的智能体选择向上移动而非向右，以降低误入-1 状态的概率。

当 $R(s) = -0.04$ 时，步惩罚增大，智能体更倾向于选择最短路径到达 $+1$ 终止状态。此时路径长度的代价超过了规避风险的收益，因此策略变得更加激进。这一设置产生的策略与经典 Grid World 问题的最优解一致。

值迭代和策略迭代虽然实现方式不同，但收敛到相同的最优策略。值迭代直接对状态值函数进行 Bellman 最优更新，而策略迭代采用策略评估与策略改进交替进行的方式。

2 免模型算法分析

MC 和 Q-Learning 均为无模型方法，无需已知环境的状态转移概率，通过与环境交互进行学习。

Q-Learning 采用时序差分更新，每执行一步即可更新 Q 值。其更新公式为 $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$，其中 $\max_{a'} Q(s',a')$ 项使得算法具有 off-policy 特性，即学习的是最优策略而非当前执行的策略。实验结果显示 Q-Learning 收敛较快且曲线相对平稳。

Monte Carlo ES 需要等待完整 episode 结束后才能更新 Q 值，采用首次访问的方式计算状态-动作对的回报均值。”探索起点” 机制通过随机选择初始状态和动作来保证所有状态-动作对都能被充分访问。由于回报 G 的方差较大且更新频率较低，MC 的学习曲线波动明显大于 Q-Learning。

两种算法最终均能收敛到接近最优的策略。Q-Learning 在本问题中表现更优，主要原因是 Grid World 的 episode 较短，单步更新的效率优势明显。

3 收敛性与时间分析

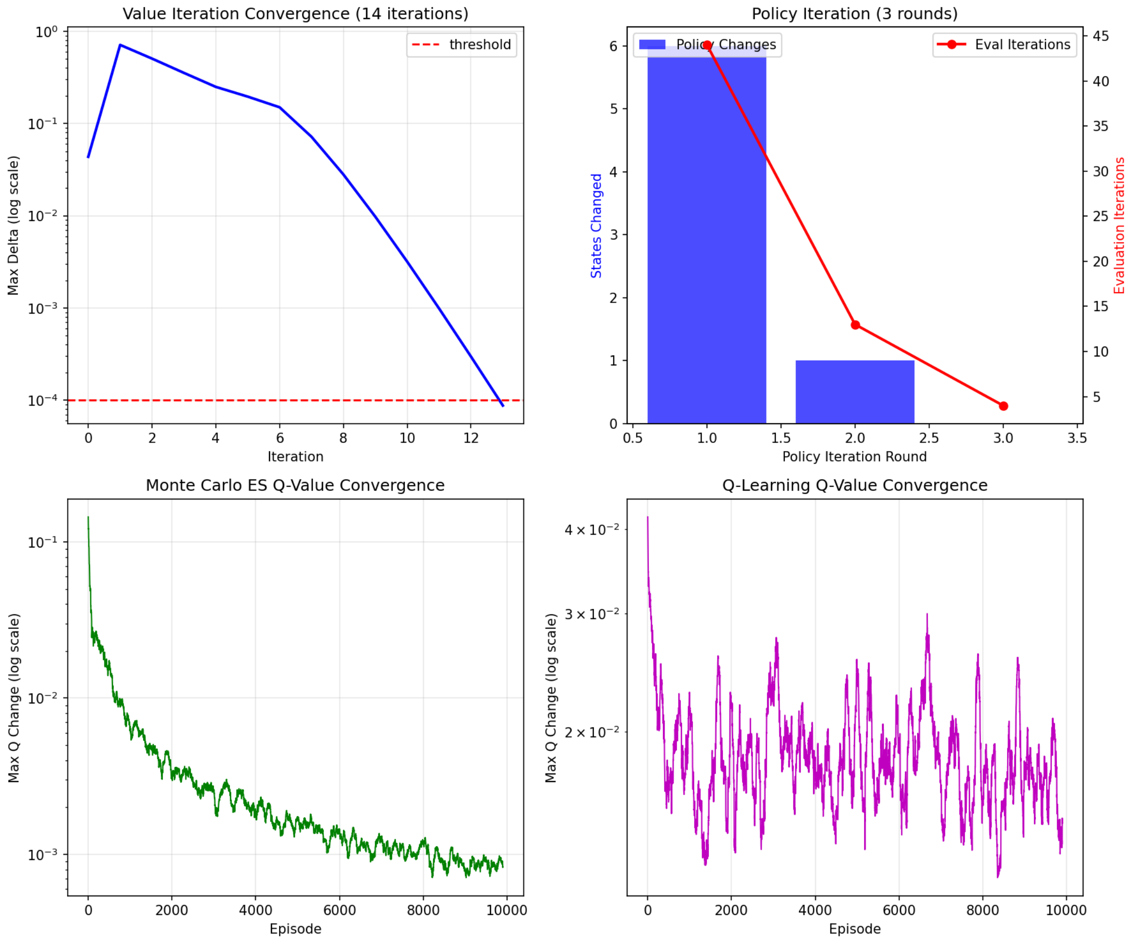

图 8: 四种算法的收敛过程与时间对比

从收敛图可以分析四种算法的特性。

值迭代的 delta 值（状态值最大变化量）呈指数下降趋势，约 20 余次迭代即可收敛至 10^{-4} 以下。这是由于 Bellman 更新算子是压缩映射，收敛速度由 γ 决定。总耗时约为数毫秒级别。

策略迭代仅需 3-4 轮即可找到最优策略。每轮包含一次完整的策略评估（需要多次迭代）和一次策略改进。由于策略空间远小于值空间，且策略改进保证单调性，策略迭代在本问题上收敛更快。

Q-Learning 的 Q 值变化在前 1000 个 episode 内快速下降并趋于稳定。单步更新机制使得信息利用效率较高。后期曲线不会完全归零，这是由于 ϵ-greedy 策略持续引入探索噪声。总耗时约为数秒。

Monte Carlo 的收敛速度明显慢于 Q-Learning，且波动更大。这是 MC 方法的固有特点：必须等待 episode 结束才能更新，且回报 G 的方差较大。从时间曲线可以看出，MC 的计算开销与 Q-Learning 相当，但收敛所需的 episode 数更多。

四、 实验总结

本次实验实现了值迭代、策略迭代、蒙特卡洛和 Q-Learning 四种强化学习算法，验证了基于模型与无模型方法的核心思想。动态规划方法在已知模型时能够精确求解最优策略，不同奖赏设置显著影响智能体行为；无模型方法中，Q-Learning 凭借单步更新特性展现出更快的收敛速度和更稳定的学习曲线。实验过程中，边界条件处理、状态转移概率计算和探索策略设计是主要实现难点。

A 代码实现

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
import time

# 固定随机种子
np.random.seed(42)
random.seed(42)


class GridWorld:  # 3x4网格世界
    def __init__(self, step_reward=-0.04):
        self.rows = 3
        self.cols = 4
        self.start = (0, 0)
        self.wall = (1, 1)
        self.terminals = {(3, 2): 1.0, (3, 1): -1.0}
        self.step_reward = step_reward
        self.gamma = 0.9
        self.actions = [(0, 1), (0, -1), (-1, 0), (1, 0)]
        self.action_names = ['^', 'v', '<', '>']

    def is_terminal(self, state):
        return state in self.terminals

    def get_reward(self, state):
        if state in self.terminals:
            return self.terminals[state]
        return self.step_reward

    def move(self, state, action):
        new_x = state[0] + action[0]
        new_y = state[1] + action[1]
        if new_x < 0 or new_x >= self.cols or \
           new_y < 0 or new_y >= self.rows or \
           (new_x, new_y) == self.wall:
```

```python
            return state
        return (new_x, new_y)

    def get_transition_probs(self, state, action_idx):
        probs = []
        action = self.actions[action_idx]
        probs.append((0.8, self.move(state, action)))
        if action[0] == 0:
            probs.append((0.1, self.move(state, (-1, 0))))
            probs.append((0.1, self.move(state, (1, 0))))
        else:
            probs.append((0.1, self.move(state, (0, 1))))
            probs.append((0.1, self.move(state, (0, -1))))
        return probs

    def all_states(self):
        states = []
        for x in range(self.cols):
            for y in range(self.rows):
                if (x, y) != self.wall:
                    states.append((x, y))
        return states


def value_iteration(env, theta=1e-4):  # 值迭代
    V = {s: 0.0 for s in env.all_states()}
    delta_history = []
    time_history = []  # 累计时间
    start_time = time.time()

    while True:
        delta = 0
        for s in env.all_states():
            if env.is_terminal(s):
                V[s] = env.get_reward(s)
                continue
            v = V[s]
            max_val = float('-inf')
            for a_idx in range(len(env.actions)):
                val = 0
                for prob, next_s in env.get_transition_probs(s, a_idx):
                    val += prob * V[next_s]
                max_val = max(max_val, val)
            V[s] = env.get_reward(s) + env.gamma * max_val
            delta = max(delta, abs(v - V[s]))

        delta_history.append(delta)
        time_history.append((time.time() - start_time) * 1000)  # ms
```

```python
        if delta < theta:
            break

    # 提取策略
    policy = {}
    for s in env.all_states():
        if env.is_terminal(s):
            policy[s] = None
            continue
        best_a, best_val = -1, float('-inf')
        for a_idx in range(len(env.actions)):
            val = sum(p * V[ns] for p, ns in env.get_transition_probs(s, a_idx))
            if val > best_val:
                best_val, best_a = val, a_idx
        policy[s] = best_a
    return V, policy, delta_history, time_history


def policy_evaluation(policy, env, V, theta=1e-4):  # 策略评估
    iterations = 0
    while True:
        delta = 0
        for s in env.all_states():
            if env.is_terminal(s):
                continue
            v = V[s]
            val = sum(p * V[ns] for p, ns in env.get_transition_probs(s, policy[s]))
            V[s] = env.get_reward(s) + env.gamma * val
            delta = max(delta, abs(v - V[s]))
        iterations += 1
        if delta < theta:
            break
    return V, iterations


def policy_iteration(env):  # 策略迭代
    policy = {s: 0 for s in env.all_states() if not env.is_terminal(s)}
    for s in env.terminals:
        policy[s] = None
    V = {s: 0.0 for s in env.all_states()}
    for s in env.terminals:
        V[s] = env.get_reward(s)

    policy_changes = []
    eval_iterations = []
    time_history = []  # 累计时间
    start_time = time.time()

```

```
    while True:
        V, iters = policy_evaluation(policy, env, V)
        eval_iterations.append(iters)

        changes = 0
        for s in env.all_states():
            if env.is_terminal(s):
                continue
            old_action = policy[s]
            best_a, best_val = -1, float('-inf')
            for a_idx in range(len(env.actions)):
                val = sum(p * V[ns] for p, ns in env.get_transition_probs(s, a_idx))
                if val > best_val:
                    best_val, best_a = val, a_idx
            policy[s] = best_a
            if old_action != best_a:
                changes += 1

        policy_changes.append(changes)
        time_history.append((time.time() - start_time) * 1000)
        if changes == 0:
            break
    return V, policy, policy_changes, eval_iterations, time_history


def monte_carlo_es(env, episodes=10000):  # MC探索起点
    Q = {(s, a): 0.0 for s in env.all_states() for a in range(len(env.actions))}
    returns = {(s, a): [] for s in env.all_states() for a in range(len(env.actions))}
    policy = {s: np.random.choice(len(env.actions)) for s in env.all_states() if not env.
     is_terminal(s)}

    non_terminal = [s for s in env.all_states() if not env.is_terminal(s)]
    history_returns, q_changes, time_history = [], [], []
    start_time = time.time()

    for ep in range(episodes):
        episode = []
        curr_state = random.choice(non_terminal)
        curr_action = np.random.choice(len(env.actions))

        steps = 0
        while not env.is_terminal(curr_state) and steps < 1000:
            probs = env.get_transition_probs(curr_state, curr_action)
            r = random.random()
            cum = 0
            for p, ns in probs:
                cum += p
                if r <= cum:
```

```
                    next_state = ns
                    break
            reward = env.get_reward(next_state)
            episode.append((curr_state, curr_action, reward))
            curr_state = next_state
            if env.is_terminal(curr_state):
                break
            curr_action = policy[curr_state]
            steps += 1

        visited, G, max_change = set(), 0, 0
        for i in range(len(episode) - 1, -1, -1):
            s, a, r = episode[i]
            G = r + env.gamma * G
            if (s, a) not in visited:
                visited.add((s, a))
                old_q = Q[(s, a)]
                returns[(s, a)].append(G)
                Q[(s, a)] = np.mean(returns[(s, a)])
                max_change = max(max_change, abs(Q[(s, a)] - old_q))
                policy[s] = max(range(len(env.actions)), key=lambda x: Q[(s, x)])

        history_returns.append(sum(r for _, _, r in episode))
        q_changes.append(max_change)
        time_history.append((time.time() - start_time) * 1000)

    final_policy = {s: max(range(len(env.actions)), key=lambda a: Q[(s, a)]) for s in
    non_terminal}
    return Q, final_policy, history_returns, q_changes, time_history


def q_learning(env, episodes=10000, alpha=0.1, epsilon=0.1):  # Q-Learning
    Q = {(s, a): 0.0 for s in env.all_states() for a in range(len(env.actions))}
    non_terminal = [s for s in env.all_states() if not env.is_terminal(s)]
    history_rewards, q_changes, time_history = [], [], []
    start_time = time.time()

    for ep in range(episodes):
        state = env.start
        total_reward, max_change = 0, 0
        steps = 0

        while not env.is_terminal(state) and steps < 1000:
            if random.random() < epsilon:
                action = np.random.choice(len(env.actions))
            else:
                action = max(range(len(env.actions)), key=lambda a: Q[(state, a)])

```

```
            probs = env.get_transition_probs(state, action)
            r = random.random()
            cum = 0
            for p, ns in probs:
                cum += p
                if r <= cum:
                    next_state = ns
                    break

            reward = env.get_reward(next_state)
            if env.is_terminal(next_state):
                target = reward
            else:
                target = reward + env.gamma * max(Q[(next_state, a)] for a in range(len(env.actions)))

            old_q = Q[(state, action)]
            Q[(state, action)] += alpha * (target - old_q)
            max_change = max(max_change, abs(Q[(state, action)] - old_q))

            total_reward += reward
            state = next_state
            steps += 1

        history_rewards.append(total_reward)
        q_changes.append(max_change)
        time_history.append((time.time() - start_time) * 1000)

    policy = {s: max(range(len(env.actions)), key=lambda a: Q[(s, a)]) for s in non_terminal}
    return Q, policy, history_rewards, q_changes, time_history


def plot_grid_policy(policy, V, env, title, filename):
    plt.figure(figsize=(8, 6))
    grid_val = np.zeros((3, 4))
    for s, val in V.items():
        grid_val[2 - s[1], s[0]] = val
    for s, reward in env.terminals.items():
        grid_val[2 - s[1], s[0]] = reward
    sns.heatmap(grid_val, annot=True, cmap="YlGnBu", cbar=True, fmt='.2f')

    arrow_map = {0: '↑', 1: '↓', 2: '←', 3: '→'}
    for s, action in policy.items():
        if action is not None:
            plt.text(s[0] + 0.5, 2 - s[1] + 0.25, arrow_map[action],
                     ha='center', va='center', fontsize=18, color='red', fontweight='bold')
    plt.text(1.5, 1.5, 'WALL', ha='center', va='center', color='black', fontsize=12,
```

```
    backgroundcolor='gray')
    for s, reward in env.terminals.items():
        plt.text(s[0] + 0.5, 2 - s[1] + 0.25, '+1' if reward > 0 else '-1',
                 ha='center', va='center', fontsize=14, color='red', fontweight='bold')
    plt.title(title)
    plt.savefig(filename)
    plt.close()


def plot_rewards(mc_hist, q_hist, filename):
    plt.figure(figsize=(10, 5))
    window = 100
    plt.plot(np.convolve(mc_hist, np.ones(window)/window, mode='valid'), label='Monte Carlo
    ES')
    plt.plot(np.convolve(q_hist, np.ones(window)/window, mode='valid'), label='Q-Learning')
    plt.xlabel('Episodes')
    plt.ylabel('Episode Return')
    plt.title('Learning Performance Comparison')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.savefig(filename)
    plt.close()


def plot_convergence(vi_delta, vi_time, pi_changes, pi_evals, pi_time,
                     mc_changes, mc_time, q_changes, q_time, filename):
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # 值迭代
    ax1 = axes[0, 0]
    ax1.semilogy(vi_delta, 'b-', linewidth=2, label='Delta')
    ax1.axhline(y=1e-4, color='r', linestyle='--', alpha=0.5)
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Max Delta (log)', color='blue')
    ax1.set_title(f'Value Iteration ({len(vi_delta)} iters, {vi_time[-1]:.1f}ms)')
    ax1.grid(True, alpha=0.3)
    ax1_t = ax1.twinx()
    ax1_t.plot(vi_time, 'g--', linewidth=1, label='Time')
    ax1_t.set_ylabel('Cumulative Time (ms)', color='green')

    # 策略迭代
    ax2 = axes[0, 1]
    x = range(1, len(pi_changes) + 1)
    ax2.bar(x, pi_changes, alpha=0.7, color='blue', label='Policy Changes')
    ax2.set_xlabel('Round')
    ax2.set_ylabel('States Changed', color='blue')
    ax2.set_title(f'Policy Iteration ({len(pi_changes)} rounds, {pi_time[-1]:.1f}ms)')
    ax2_t = ax2.twinx()
```

```python
    ax2_t.plot(x, pi_time, 'g-o', linewidth=2, label='Time')
    ax2_t.set_ylabel('Cumulative Time (ms)', color='green')

    # MC
    ax3 = axes[1, 0]
    window = 100
    mc_smooth = np.convolve(mc_changes, np.ones(window)/window, mode='valid')
    ax3.semilogy(mc_smooth, 'b-', linewidth=1)
    ax3.set_xlabel('Episode')
    ax3.set_ylabel('Max Q Change (log)', color='blue')
    ax3.set_title(f'Monte Carlo ES ({mc_time[-1]/1000:.2f}s)')
    ax3.grid(True, alpha=0.3)
    ax3_t = ax3.twinx()
    # 每1000个episode画一个时间点
    time_x = list(range(0, len(mc_time), 1000))
    time_y = [mc_time[i]/1000 for i in time_x]
    ax3_t.plot(time_x, time_y, 'g--', linewidth=1)
    ax3_t.set_ylabel('Time (s)', color='green')

    # Q-Learning
    ax4 = axes[1, 1]
    q_smooth = np.convolve(q_changes, np.ones(window)/window, mode='valid')
    ax4.semilogy(q_smooth, 'b-', linewidth=1)
    ax4.set_xlabel('Episode')
    ax4.set_ylabel('Max Q Change (log)', color='blue')
    ax4.set_title(f'Q-Learning ({q_time[-1]/1000:.2f}s)')
    ax4.grid(True, alpha=0.3)
    ax4_t = ax4.twinx()
    time_x = list(range(0, len(q_time), 1000))
    time_y = [q_time[i]/1000 for i in time_x]
    ax4_t.plot(time_x, time_y, 'g--', linewidth=1)
    ax4_t.set_ylabel('Time (s)', color='green')

    plt.tight_layout()
    plt.savefig(filename, dpi=150)
    plt.close()


if __name__ == "__main__":
    print("Running Value Iteration and Policy Iteration...")
    env = GridWorld(step_reward=-0.04)

    V_vi, policy_vi, vi_delta, vi_time = value_iteration(env)
    print(f"  VI: {len(vi_delta)} iterations, {vi_time[-1]:.2f}ms")

    V_pi, policy_pi, pi_changes, pi_evals, pi_time = policy_iteration(env)
    print(f"  PI: {len(pi_changes)} rounds, {pi_time[-1]:.2f}ms")
```

```python
    # 不同R(s)的策略图
    for r in [0.01, -0.01, -0.04]:
        env_r = GridWorld(step_reward=r)
        V, policy, _, _ = value_iteration(env_r)
        plot_grid_policy(policy, V, env_r, f"Value Iteration Policy (R={r})", f"policy_vi_{r
    }.png")

    print("Running MC and Q-Learning...")
    env = GridWorld(step_reward=-0.04)

    Q_mc, policy_mc, mc_hist, mc_changes, mc_time = monte_carlo_es(env, episodes=10000)
    print(f"  MC: 10000 episodes, {mc_time[-1]/1000:.2f}s")

    Q_q, policy_q, q_hist, q_changes, q_time = q_learning(env, episodes=10000)
    print(f"  Q-Learning: 10000 episodes, {q_time[-1]/1000:.2f}s")

    # 画策略图
    V_mc = {s: max(Q_mc.get((s, a), 0) for a in range(4)) for s in env.all_states() if not
    env.is_terminal(s)}
    V_q = {s: max(Q_q.get((s, a), 0) for a in range(4)) for s in env.all_states() if not env
    .is_terminal(s)}

    plot_grid_policy(policy_mc, V_mc, env, "Monte Carlo Policy", "policy_mc.png")
    plot_grid_policy(policy_q, V_q, env, "Q-Learning Policy", "policy_q.png")
    plot_rewards(mc_hist, q_hist, "learning_comparison.png")

    # 收敛过程对比图
    plot_convergence(vi_delta, vi_time, pi_changes, pi_evals, pi_time,
                     mc_changes, mc_time, q_changes, q_time, "convergence_comparison.png")

    print("Done.")
```