**作业一 杰克租车问题**

**一 问题描述**

杰克管理一家有两个地点的租车公司。每一天，一些用户会到一个地 点租车。如果杰克有可用的汽车，便会将其租出，并从全国总公司那里获得10美 元的收益。如果他在那个地点没有汽车，便会失去这一业务。租出去的汽车在还 车的第二天变得可用。为了保证每辆车在需要的地方使用，杰克在夜间在两个地 点之间移动车辆，移动每辆车的代价为2美元。我们假设每个地点租车与还车的 数量是一个泊松随机变量，即数量为n的概率为 。假设租车的期望在两个 地点分别为3和4，而还车的期望分别为3和2。为了简化问题，我们假设任何一个 地点有不超过20辆车，并且每天最多移动5辆车。折扣率为0.9。

已知：

状态空间：2个地点，每个地点最多20辆车供租赁 ；

行为空间：每天下班后最多转移5辆车从一处到另一处；

即时奖励：每租出1辆车奖励10元，必须是有车可租的情况。不考虑在两地转移车辆的支出；

转移概率：求租和归还是随机的，但是满足泊松分布 。第一处租赁点平均每天租车 请求3次，归还3次；第二处租赁点平均每天租车4次，归还2次；

衰减系数 γ：0.9；

问题：怎样的策略是最优策略？

**二 实验环境**

编程语言：Python

操作系统：Windows

编辑器：Visual Studio Code

**三 具体实现**

为了完成该题目，使用了强化学习中的动态规划，实现策略迭代算法来解决杰克租车问题。以下是具体的程序流程：

* 导入必要的库

import math

import numpy as np

* 初始化参数：

# 参数设置

MAX\_CARS = 20       # 每个地点的最多汽车数量

MAX\_MOVE = 5        # 每次最多移动汽车的数量

DISCOUNT = 0.9      # 折扣率

RENT\_REWARD = 10.0  # 租车收益

MOVE\_COST = 2.0     # 移动成本

# 租车和还车的期望值

rental\_lambda = [3, 4]

return\_lambda = [3, 2]

# 初始化状态值和策略矩阵

state\_values = np.zeros(shape=(MAX\_CARS + 1, MAX\_CARS + 1), dtype=np.float32)

policy = np.zeros(shape=(MAX\_CARS + 1, MAX\_CARS + 1), dtype=int)

# 定义动作空间

actions = np.arange(-MAX\_MOVE, MAX\_MOVE + 1)

这些参数的定义主要用于配置杰克租车问题的环境和算法参数。MAX\_CARS 和 MAX\_MOVE 分别表示每个地点的最大汽车数量和每次最大移动汽车的数量。DISCOUNT 是折扣率，用于计算未来奖励的折扣值。RENT\_REWARD 和 MOVE\_COST 分别表示租车收益和移动汽车的成本。rental\_lambda 和 return\_lambda 是两个地点租车和还车的期望值，用于描述每个地点租车和还车的泊松分布参数。最后，state\_values 和 policy 是用于存储状态值和策略的矩阵。actions 定义了动作空间，表示可选的移动汽车的动作范围，负值表示把车从第二地点到第一地点移动，正值表示把车从第一地点到第二地点移动。这些参数的设定旨在提供一个合适的环境和算法设置，以解决杰克租车问题。

* 定义状态转移函数：

# 状态转移函数

def poisson\_dis(lam, n):

    return (lam\*\*n / math.factorial(n)) \* np.exp(-lam)

poisson\_dis 函数定义了泊松分布的概率质量函数（Probability Mass Function，PMF）。其中，lam 是泊松分布的平均值，表示在单位时间内发生某事件的期望次数，而 n 则表示实际发生的次数。该函数计算了泊松分布给定平均值和实际次数的概率值，用于描述租车和还车事件在每个地点的随机性，从而在问题建模中考虑到了这种不确定性。

* 定义动作价值函数：

# 动作价值函数

def action\_value\_function(state, action, state\_values):

    reward = 0

    first\_loc\_cars = state[0] - action

    second\_loc\_cars = state[1] + action

    for first\_loc\_rentals in range(first\_loc\_cars + 1):

        for second\_loc\_rentals in range(second\_loc\_cars + 1):

            for first\_loc\_returns in range(MAX\_CARS + 1):

                for second\_loc\_returns in range(MAX\_CARS + 1):

                    action\_value = (first\_loc\_rentals + second\_loc\_rentals) \* RENT\_REWARD

                    action\_value -= abs(action) \* MOVE\_COST

                    next\_first\_loc\_cars = min(first\_loc\_cars - first\_loc\_rentals + first\_loc\_returns,

                                              MAX\_CARS)

                    next\_second\_loc\_cars = min(second\_loc\_cars - second\_loc\_rentals + second\_loc\_returns,

                                               MAX\_CARS)

                    probability = (poisson\_dis(rental\_lambda[0], first\_loc\_rentals)

                                   \* poisson\_dis(rental\_lambda[1], second\_loc\_rentals)

                                   \* poisson\_dis(return\_lambda[0], first\_loc\_returns)

                                   \* poisson\_dis(return\_lambda[1], second\_loc\_returns))

                    reward += probability \* (

                        action\_value + DISCOUNT \* state\_values[next\_first\_loc\_cars, next\_second\_loc\_cars])

    return reward

action\_value\_function 函数定义了动作价值函数，用于计算在给定状态、动作和当前状态值矩阵的情况下，执行该动作所产生的期望回报。具体而言，函数考虑了每个地点的租车和还车情况的随机性，通过4次嵌套循环遍历所有可能的租车和还车组合，并计算对应的动作价值。该函数同时考虑了租车的奖励、移动车辆的成本以及下一状态的折扣后的状态值，以综合评估执行该动作的价值。这有助于在策略迭代中更新状态值和制定最优策略。

* Main函数（包含策略迭代和迭代日志记录）

if \_\_name\_\_ == '\_\_main\_\_':

    file = open('iteration\_log.txt', 'w')

    # 策略迭代

    iteration = 0

    while True:

        iteration += 1

        # 复制当前的状态价值和策略矩阵

        new\_state\_values = np.copy(state\_values)

        new\_policy = np.copy(policy)

        # 遍历每个状态

        for first\_loc\_cars in range(MAX\_CARS + 1):

            for second\_loc\_cars in range(MAX\_CARS + 1):

                state = (first\_loc\_cars, second\_loc\_cars)

                # 当前状态下动作的价值列表

                action\_value\_list = []

                # 遍历所有可能的动作

                for action in actions:

                    # 定义合理的动作

                    action\_doable = ((action >= 0 and (first\_loc\_cars >= action) and

                                      second\_loc\_cars + action <= MAX\_CARS) or

                                     (action < 0 and second\_loc\_cars >= abs(action) and

                                      first\_loc\_cars + abs(action) <= MAX\_CARS))

                    if action\_doable:

                        action\_value\_list.append(action\_value\_function(state, action, new\_state\_values))

                    else:

                        action\_value\_list.append(-np.inf)

                #选择最大回报的动作作为新的状态值和动作

                new\_state\_values[first\_loc\_cars, second\_loc\_cars] = max(action\_value\_list)

                new\_policy[first\_loc\_cars, second\_loc\_cars] = np.argmax(action\_value\_list) - 5

                # print(f"iteration{iteration} state{state}")

        # 计算新状态值和旧状态值的总差值

        delta = np.sum(np.abs(new\_state\_values - state\_values))

        policy\_stable = (policy == new\_policy).all()

        # 将新状态值和策略矩阵复制到之前的状态值和策略矩阵

        state\_values = new\_state\_values

        policy = new\_policy

        # 输出该迭代的信息

        text = (f"iteration:{iteration}\ndelta:{delta}\nupdated state values:\n{state\_values}\n"

                f"updated policy:\n{policy}\n")

        print(text)

        # 记录该迭代的信息到txt文件

        file.write(text)

        file.write('--------------------------------------------------------------------------------------------\n')

        # 如果状态值收敛并且策略稳定，则结束迭代

        if delta < 1e-4 or policy\_stable:

            print("iteration finished! state values convergence")

            break

    file.close()

该部分代码是实现策略迭代的主要循环。在 \_\_main\_\_ 函数中，首先打开一个名为 'iteration\_log.txt' 的文件以记录迭代的信息。然后通过一个 while True 循环进行策略迭代，其中每次迭代都会更新状态值和策略矩阵。在每次迭代中，首先复制当前的状态值和策略矩阵，然后遍历所有可能的状态。对于每个状态，在每个状态中再遍历动作空间，使用动作价值函数计算每个动作的价值，并选择使得价值最大的动作作为新的状态值和策略。这里的动作空间在 actions 中定义，考虑了每次最多移动汽车的数量。

在更新完所有状态的状态值和策略后，计算新状态值和旧状态值的总差值delta，并检查新策略和旧策略是否一样。如果状态值的变化delta小于阈值（1e-4）或策略稳定（即新策略与旧策略相同），则结束迭代。在每次迭代中，将迭代的信息输出到 'iteration\_log.txt' 文件中，包括迭代次数、状态值的变化delta、更新后的状态值和策略矩阵。

最终，关闭记录信息的文件，完成整个策略迭代的过程。

**四 实验结果**

根据迭代日志记录文件 'iteration\_log.txt'，在经过10次迭代后，策略已经稳定。最后的策略矩阵如下：

[[ 0 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5]

[ 1 0 0 -1 -1 -2 -2 -2 -3 -3 -4 -4 -5 -5 -5 -5 -5 -5 -5 -5 -5]

[ 1 1 0 0 -1 -1 -1 -2 -2 -3 -3 -4 -4 -4 -5 -5 -5 -5 -5 -5 -5]

[ 2 1 1 0 0 0 -1 -1 -2 -2 -3 -3 -3 -4 -4 -4 -4 -5 -5 -5 -5]

[ 2 2 1 1 0 0 0 -1 -1 -2 -2 -2 -3 -3 -3 -3 -4 -4 -4 -4 -4]

[ 3 2 2 1 1 0 0 0 -1 -1 -1 -2 -2 -2 -2 -3 -3 -3 -3 -3 -3]

[ 3 3 2 2 1 1 1 0 0 0 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -3]

[ 4 3 3 2 2 2 1 1 0 0 0 0 0 -1 -1 -1 -1 -1 -1 -2 -2]

[ 4 4 3 3 3 2 2 1 0 0 0 0 0 0 0 0 0 0 -1 -1 -1]

[ 5 4 4 4 3 3 2 1 1 0 0 0 0 0 0 0 0 0 0 0 -1]

[ 5 5 5 4 4 3 2 2 1 1 0 0 0 0 0 0 0 0 0 0 0]

[ 5 5 5 5 4 3 3 2 2 1 0 0 0 0 0 0 0 0 0 0 0]

[ 5 5 5 5 4 4 3 3 2 1 1 0 0 0 0 0 0 0 0 0 0]

[ 5 5 5 5 5 4 4 3 2 2 1 0 0 0 0 0 0 0 0 0 0]

[ 5 5 5 5 5 5 4 3 3 2 1 0 0 0 0 0 0 0 0 0 0]

[ 5 5 5 5 5 5 4 4 3 2 1 1 0 0 0 0 0 0 0 0 0]

[ 5 5 5 5 5 5 5 4 3 2 2 1 1 0 0 0 0 0 0 0 0]

[ 5 5 5 5 5 5 5 4 3 3 2 2 1 1 1 1 0 0 0 0 0]

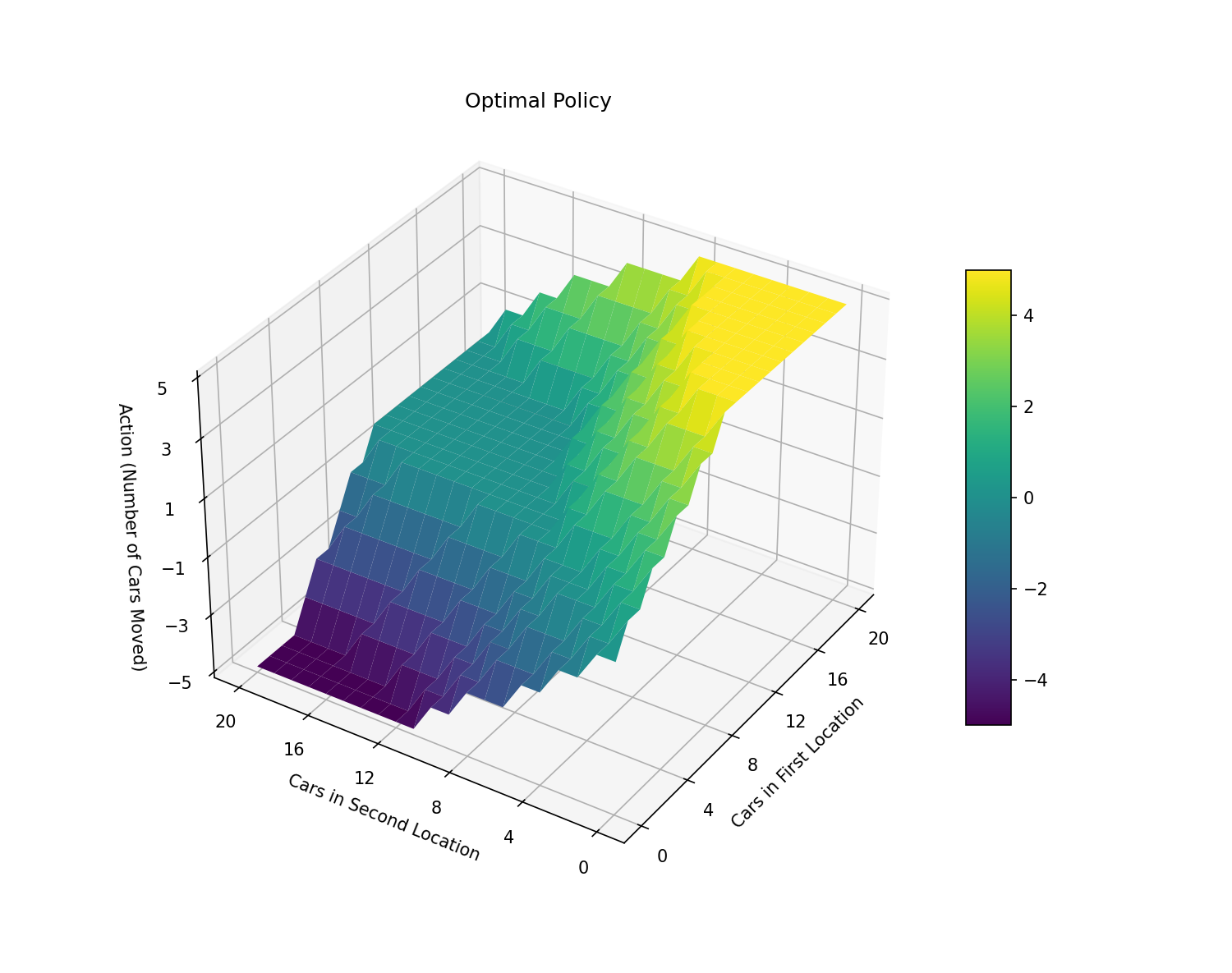
[ 5 5 5 5 5 5 5 4 4 3 3 2 2 2 2 1 1 1 0 0 0]

[ 5 5 5 5 5 5 5 5 4 4 3 3 3 3 2 2 2 1 1 0 0]

[ 5 5 5 5 5 5 5 5 5 4 4 4 4 3 3 3 2 2 1 1 0]]

其中行号表示第一地点的车数量，列号表示第二地点的车数量。因为第二地点的租车期望值比第一地点的租车期望值多，表明第二个地点租车的人比较多，且第二地点的还车期望值比第一地点的还车期望值多，表明第二地点被还的车比较少，导致第二地点比第一地点的车会更快被租完。因此我们可以看到在策略稳定的矩阵中，汽车的移动更偏向于从第一个位置到第二个位置的移动。

我们可以通过matplotlib将这个稳定策略矩阵以三维形式可视化，具体代码可以看压缩包里的visualize.py文件。得到的图像如下：



其中x轴为第一地点的车数量，y轴为第二地点的车数量，z轴为x轴和z轴组合的状态的最优策略。在z轴，负值表示把车从第二地点到第一地点移动，正值表示把车从第一地点到第二地点移动。

**四 实验总结**

在这个实验中，使用了动态规划中的策略迭代方法解决了杰克租车问题。通过对状态值和策略矩阵的迭代更新，最终获得了一个稳定的最优策略矩阵，其中每个状态对应的动作表示了在不同汽车分布情况下的最优移动汽车决策。通过观察最终的策略矩阵，我们可以发现汽车的移动更偏向于从第一个位置到第二个位置的移动，这与问题中第二个地点的车更快被租完相吻合。最后，通过matplotlib将最终的稳定策略矩阵以三维形式可视化，直观展示了最优策略在不同状态下的车辆移动方向。