作业 2：DP_rent_car 实验报告

人工智能 91 卢佳源 2191121196

一、 实验目的：

a) 理解动态规划原理（策略迭代和价值迭代）和实现过程；
b) 编写程序，解决杰克租车问题。

二、 问题重述：杰克租车问题

a) 两个租车地点，每地最多 20 辆车，每天最多移动 5 辆车；
b) 租出一辆车，获得 10 美元；移动一辆车，花费 2 美元；
c) 每个地点的租车和还车数量服从泊松分布，其中参数设置如下：

	地点 1	地点 2
租车参数 λ	3	4
还车参数σ	3	2

租出的车辆在还车后的第二天可以使用。
d) 折扣率为 0.9；
e) 该过程描述为持续的有限 MDP：
   i. 动作：在两地移动的车辆数；
   ii. 状态：每天两地的车辆数；

三、 实验环境：

a) IDE：VSCode，Python-3.9.7
b) 编程语言：Python；
c) 文件路径：C:\Users\jiayuan lu\OneDrive - MSRA\桌面\大三下\RL\作业 2 DP_rent_car\DP.ipynb

四、 实验原理和思路：

a) 计算每个状态下 S 执行一种动作 A 后得到的新的状态对应的价值：
   i. 统计动作 A 中包含“移动车辆”的次数，再乘上移动一辆车需要花费的代价，作为该状态 S 下执行动作 A 所花费的总代价；
   ii. 之后考虑动作 A 带来的租出部分的即时奖励：
      1. 执行动作 A 后，两地的车辆数会发生一定的改变，因此两地对于能够租出的车辆的能力也发生了变化，我们需要统计执行动作 A 后两地可以租出的车辆数，其中有若干个约束条件需要遵守：
         a) 两地各自的车辆数不得超过 20 辆，即当前的车辆数要取状态 S 下原始的车辆数和动作 A 移动的车辆数的总和，与单地最多车辆数 20 的

最小值；

b) 两地决定租出的车辆数取决于当前各地的车辆数和当前出租的需求量的最小值；

c) 租出后该地的车辆数要减去 b)中两地决定租出的车辆数作为更新状态后两地的车辆数；

2. 将两地本次租出的车辆数的总和，乘上租出一辆车获得的利润，作为该状态下执行动作 A 获得的租出部分的即时奖励；

iii. 接着考虑动作 A 带来的还车部分的即时奖励：

1. 在租出车辆后，我们得到了 ii 中更新后的两地车辆数，在此基础上，按照泊松分布考虑还车的数量，得到租出和还车后的当前车辆数；

2. 将状态转移概率定义为租出和还车的概率乘积（因为两者是条件独立的），然后按照状态价值函数的定义计算当前状态的执行动作 A 后的即时奖励；

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

iv. 该函数最终返回从当前状态开始执行动作后的期望累计奖励作为该状态的价值。

b) 策略评估过程：

i. 根据 a)中的状态价值函数，计算每一个专状态对应的价值，并计算更新后的状态价值和更新前的状态价值的最大差异，若该最大差异小于一个事先定义的小数，则停止策略评估的循环过程，否则一直循环直到更新前后状态价值的差异足够小为止。

c) 策略改进过程：

i. 首先定义一个 bool 类型的变量——policy-stable，并赋值为 True；

ii. 对每一个状态，将当前的策略赋给一个存储旧的动作的数组 old-action 中，并利用 A 中的状态价值函数来计算出最大状态价值对应的动作，并将该动作更新原始的策略序列，如果旧的动作数组和这个新的策略序列数组不相同，则 policy-stable=False，重新循环；直到一个新的策略和上次更新的策略完全相同时，policy-stable=True，停止循环并输出更新后的价值和策略，再次进入 b)中的策略评估过程。

d) 价值评价过程：

i. 结合了策略改进和截断策略评估，循环过程与策略评估的过程类似，唯一的不同是每次选用状态价值函数的最大值来更新状态价值。

e) 价值迭代后的最优策略：

i. 对 d)中得到的状态价值的最大值反求出对应的最优策略序列。

f) 画图：

i. 将每一次的迭代过程的策略和价值用三维热图表示出来，观察每一次迭代的变化。

五、 实验代码：

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from scipy.stats import poisson

location=[1,2]
max_car_num=20
reward_rent_onecar=10
cost_move_onecar=2
move_1day_max_car_num=5
requests_avg=[3,4]
returns_avg=[3,2]
gamma=0.9
theta=0.0001
action=np.arange(-move_1day_max_car_num,move_1day_max_car_num+1)
value = np.zeros((max_car_num+1, max_car_num+1))
policy = np.zeros(value.shape, dtype=int)
poisson_visited = dict()


def action_choice_prob(n,lamda):
    visited_prob=n*20+lamda
    if visited_prob not in poisson_visited:
        poisson_visited[visited_prob]=poisson.pmf(n,lamda)
    return poisson_visited[visited_prob]


def state_value_compute(act,state,state_value):
    state_value_update=0.0
    state_value_update-=cost_move_onecar*abs(act)
    for i in range(max_car_num+1):
        for j in range(max_car_num+1):
            request_prob=action_choice_prob(i,requests_avg[0])*action_ch
oice_prob(j,requests_avg[1])
            curr_1_car_num=min(state[0]-act,max_car_num)
            curr_2_car_num=min(state[1]+act,max_car_num)
            rent_from1_available_car_num=min(curr_1_car_num,i)
            rent_from2_available_car_num=min(curr_2_car_num,j)
            curr_reward=(rent_from1_available_car_num+rent_from2_availab
le_car_num)*reward_rent_onecar
            curr_1_car_num-=rent_from1_available_car_num
            curr_2_car_num-=rent_from2_available_car_num
            for m in range(max_car_num+1):
                for n in range(max_car_num+1):
                    return_prob=action_choice_prob(m,returns_avg[0])*act
ion_choice_prob(n,returns_avg[1])
```

```
                    curr_1_car_num_follow=min(curr_1_car_num+m,max_car_n
um)
                    curr_2_car_num_follow=min(curr_2_car_num+n,max_car_n
um)
                    prob_transition=return_prob*request_prob
                    state_value_update+=prob_transition*(curr_reward+gam
ma*state_value[curr_1_car_num_follow,curr_2_car_num_follow])
    return state_value_update


def policy_evaluation():
    while True:
        pre_value=value.copy()
        for i in range(max_car_num+1):
            for j in range(max_car_num+1):
                state_value_update=state_value_compute(policy[i,j],[i,j]
,value,return_cars=True)
                value[i,j]=state_value_update
        delta=abs(pre_value-value).max()
        print(f'delta: {delta}')
        if delta<theta:
            break
    return value


def policy_improvement():
    policy_stable=True
    for i in range(max_car_num+1):
        for j in range(max_car_num+1):
            old_action=policy[i,j]
            action_update=[]
            for k in action:
                if i>=k and -j<=k :
                    action_update.append(state_value_compute(k,[i,j],val
ue,return_cars=True))
                else:
                    action_update.append(-np.inf)
            curr_best_action=action[np.argmax(action_update)]
            policy[i,j]=curr_best_action
            if old_action!=curr_best_action and policy_stable:
                policy_stable=False
    return policy_stable,policy
```

```
def value_evalution():
    iter=0
    while True:
        pre_value=value.copy()
        for i in range(max_car_num+1):
            for j in range(max_car_num+1):
                value_update=[]
                for k in action:
                    if i>=k and -j<=k :
                        value_update.append(state_value_compute(k,[i,j],v
alue,return_cars=True))
                    else:
                        value_update.append(-np.inf)
                value_best=np.max(value_update)
                value[i,j]=value_best
        delta=abs(pre_value-value).max()
        print(f'delta: {delta}')
        iter+=1
        if delta<theta:
            break
    return value


def policy_from_optimal_value():
    for i in range(max_car_num+1):
        for j in range(max_car_num+1):
            value_update=[]
            for k in action:
                if i>=k and -j<=k :
                    value_update.append(state_value_compute(k,[i,j],valu
e,return_cars=True))
                else:
                    value_update.append(-np.inf)
            policy[i,j]=action[np.argmax(np.round(value_update,5))]
    return policy


def figure_value(value, policy):
    fig = plt.figure(figsize=(20, 20))
    ax = fig.add_subplot(121)
    ax.matshow(policy, cmap=plt.cm.bwr, vmin=-move_1day_max_car_num,
vmax=move_1day_max_car_num)
    ax.set_xticks(range(max_car_num+1))
    ax.set_yticks(range(max_car_num+1))
```

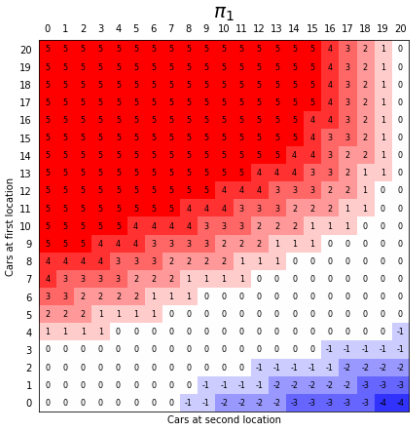
```
    ax.invert_yaxis()
    ax.set_xlabel("Cars at second location")
    ax.set_ylabel("Cars at first location")
    ax.set_title(r'$\pi_{value}$', fontsize=20)
    for x in range(max_car_num+1):
        for y in range(max_car_num+1):
            ax.text(x=x, y=y, s=int(policy.T[x, y]), va='center',
ha='center', fontsize=10)

    y, x = np.meshgrid(range(max_car_num+1), range(max_car_num+1))
    ax = fig.add_subplot(122, projection='3d')
    ax.scatter3D(y, x, value.T)
    ax.set_xlim3d(0, max_car_num)
    ax.set_ylim3d(0, max_car_num)
    ax.set_xlabel("Cars at second location")
    ax.set_ylabel("Cars at first location")
    ax.set_title('value for ' + r'$\pi_{value}$', fontsize=20)
    plt.savefig('value.png', bbox_inches='tight')


def figure_policy(value, policy, iteration):
    fig = plt.figure(figsize=(20, 20))
    ax = fig.add_subplot(121)
    ax.matshow(policy, cmap=plt.cm.bwr, vmin=-move_1day_max_car_num,
vmax=move_1day_max_car_num)
    ax.set_xticks(range(max_car_num+1))
    ax.set_yticks(range(max_car_num+1))
    ax.invert_yaxis()
    ax.set_xlabel("Cars at second location")
    ax.set_ylabel("Cars at first location")
    ax.set_title(r'$\pi_{}$'.format(iteration), fontsize=20)
    for x in range(max_car_num+1):
        for y in range(max_car_num+1):
            ax.text(x=x, y=y, s=int(policy.T[x, y]), va='center',
ha='center', fontsize=10)

    y, x = np.meshgrid(range(max_car_num+1), range(max_car_num+1))
    ax = fig.add_subplot(122, projection='3d')
    ax.scatter3D(y, x, value.T)
    ax.set_xlim3d(0, max_car_num)
    ax.set_ylim3d(0, max_car_num)
    ax.set_xlabel("Cars at second location")
    ax.set_ylabel("Cars at first location")
```

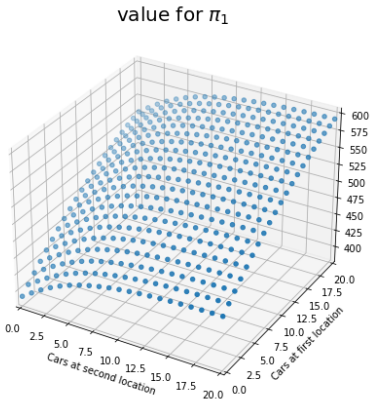
```
    ax.set_title('value for ' + r'$\pi_{}$'.format(iteration),
fontsize=20)
    plt.savefig(f'{iteration}_policy_T.png', bbox_inches='tight')


def policy_iteration():
    iter=0
    while True:
        value_update=policy_evaluation()
        stable,policy_update=policy_improvement()
        iter+=1
        print('iteration: {}, policy stable {}'.format(iter, stable))
        figure_policy(value_update,policy_update,iter)
        if stable==True:
            break


def value_interation():
    value_update=value_evalution()
    policy_update=policy_from_optimal_value()
    figure_value(value_update,policy_update)


if __name__ == '__main__':
    policy_iteration()
    value_interation()
```

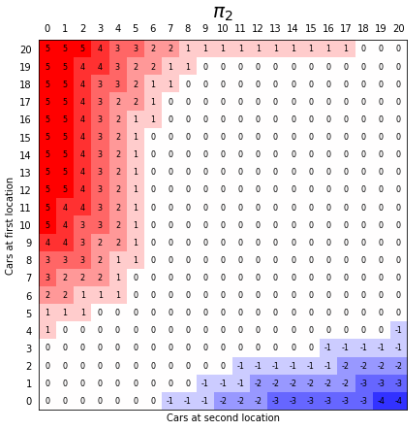
六、 实验结果：

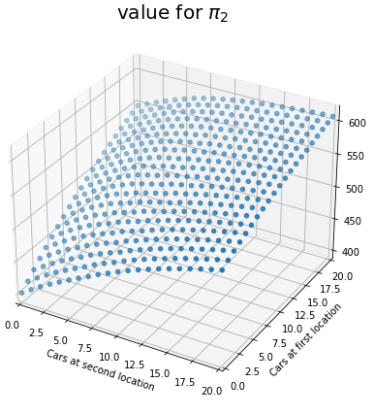
a) 实验结果:

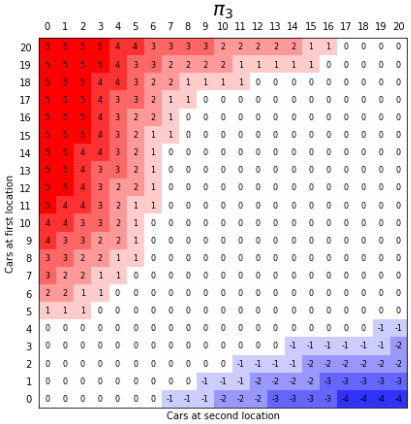
i. 策略迭代:

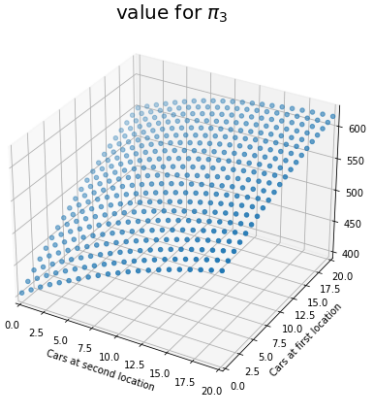


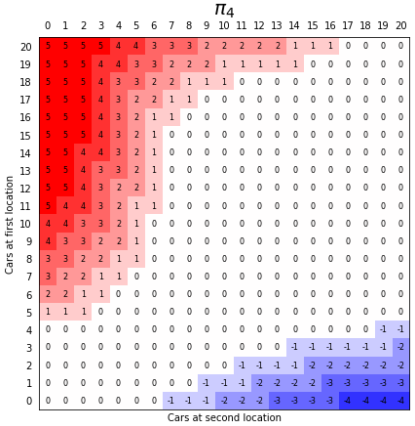


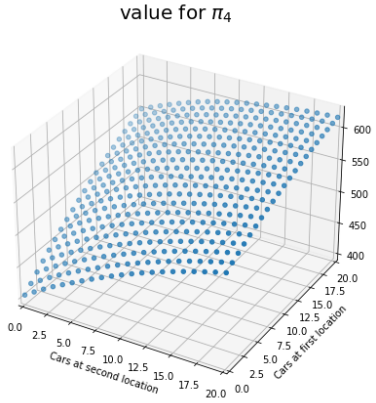

π_2
Cars at first location
Cars at second location

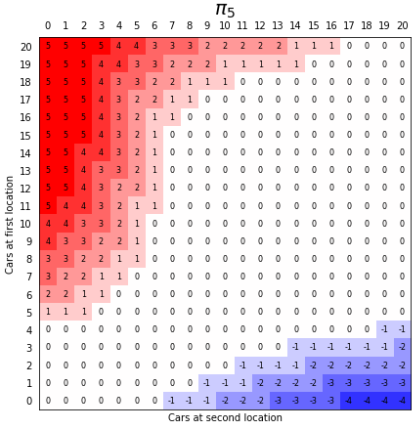

value for π_2
Cars at second location
Cars at first location

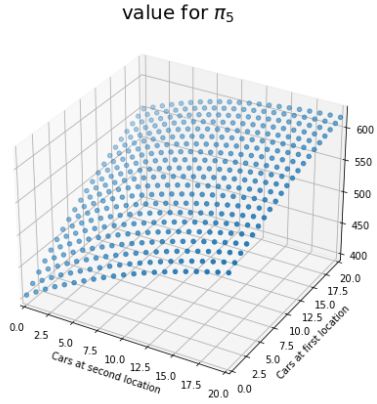

π_3
Cars at first location
Cars at second location


value for π_3
Cars at second location
Cars at first location

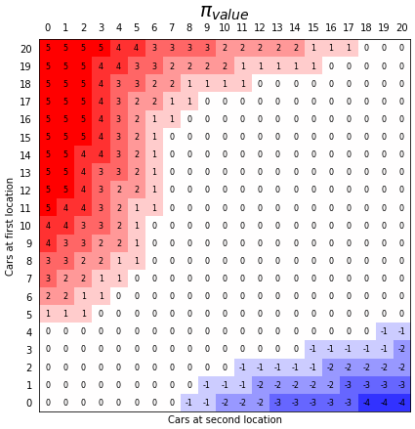

π_4
Cars at first location
Cars at second location

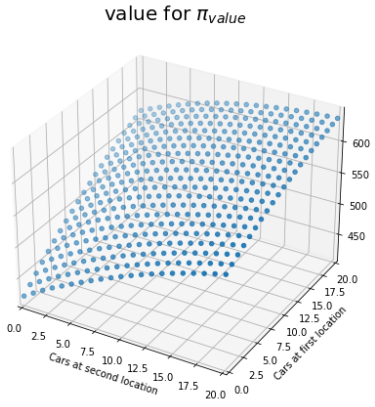

value for π_4
Cars at second location
Cars at first location


π_5
Cars at first location
Cars at second location


value for π_5
Cars at second location
Cars at first location

ii. 价值迭代：





b) 实验结果分析：

i. 由上图可知，策略迭代和价值迭代的最终结果都收敛到最优值，即最终的两种方式计算出来的三维热图是一样的；

ii. 从策略迭代的图可知，三位热图逐渐逼近最终的最优解，且在第四次和第五次的画图结果可见，当最优策略和价值不再发生变化时，迭代终止，即已经收敛到了最优策略上。

七、 实验反思：

a) 计算状态价值的函数是整个算法中时间复杂度最高的部分，达到了$O(n^4)$，使得一次运行的时间非常长，以天为单位，于是我参考了一个博客，上面是把还车数量部分直接当成了泊松分布的期望，也就省去了还车部分的两层循环，运行速度大大提高。代码如下：

```
if return_cars:
                curr_1_car_num_follow=min(curr_1_car_num+returns_avg[0],
max_car_num)
                curr_2_car_num_follow=min(curr_2_car_num+returns_avg[1],
max_car_num)
                state_value_update+=request_prob*(curr_reward+gamma*stat
e_value[curr_1_car_num_follow,curr_2_car_num_follow])
```

当然，我上面实验代码部分中的这一步，相当于 return_cars==False，即就是每一次还车都要一个个计算数量，使得算法复杂度急剧增加。另外，上面的结果图是按照没有上述优化的代码的运行结果，即就是用了三天半运行出来的结果图。

b) 策略迭代和价值迭代得到的最终结果应该是相同的，因为后者相当于前者的策略改进+截断的策略评估，即在每次迭代中更新策略，是可以证明二者是都可以收敛到最优价值函数的。