杰克出租车问题

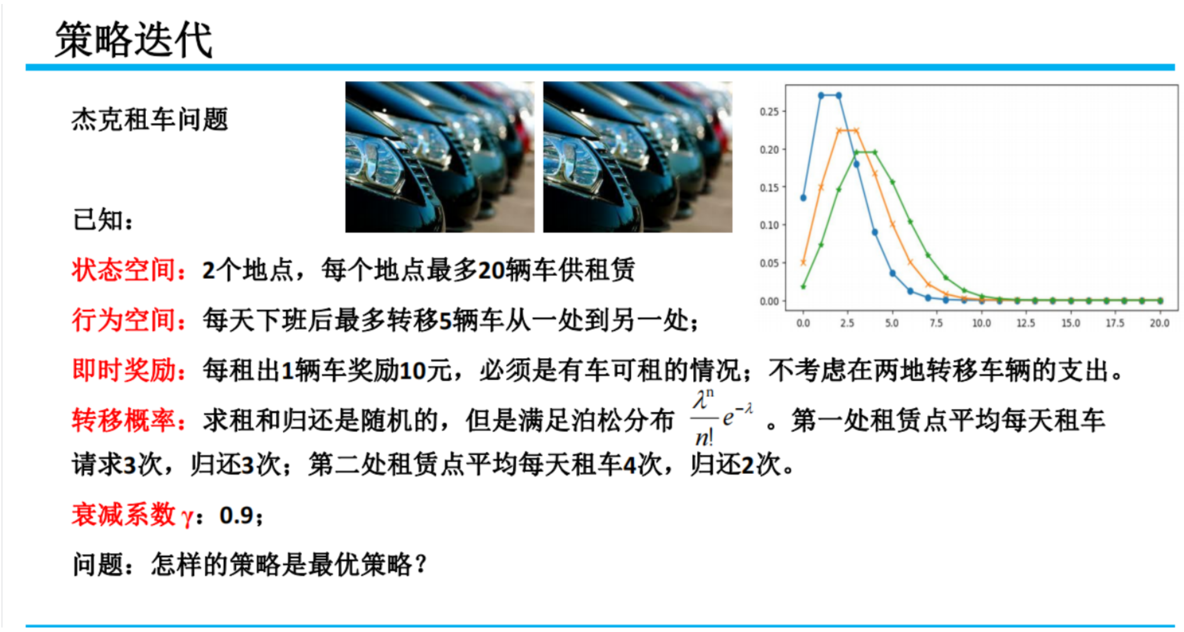
一、代码结构

```
├── ReadMe.md              // 帮助文档
├── Jack_Car_RENTAL.py     // 主函数
├── params.py              // 主要参数
├── calculate_value.py     // 更新价值函数
├── figure.py              // 绘图
|   ├── result             // 运行结果
```

二、问题描述

三、问题分析

将这个问题当作连续有限MDP，时间步骤是天数。首先要明确“状态”和“动作”是什么。 **状态**是每天结束是在每个位置剩余车子的数量，**动作**是每晚将车子在两个地点转移的净数量，然后用**动态规划**的方法解。



四、代码实现

1. **状态转移概率矩阵**

   对于租赁点一，开始时有a辆车,经过一天租车还车，车辆数为b，计算从a到b的概率tp[a,b]。

   根据泊松分布，分别计算租出r辆车和归还ret辆车**(注意要考虑实际情况)**的概率p_r,p_ret，

   则b = a-r+ret，tp[a,b] = p_r*p_ret。

```python
def trans_prob(s, loc):
    """
    :param s: 初始车辆数
    :param loc: 租赁点位置 0:第一个租赁点 1:第二个租赁点
    :return:
    """
    for r in range(0, max_car_num + 1):
        p_rent = poisson(r, request_mean[loc])  # 租出去r辆车的概率
        if p_rent < accurate:  # 精度限制
            return
        rent = min(s, r)  # 租车数不可能大于库存数
        reward[loc, s] += p_rent * rent_income * rent  # 租车收益
        for ret in range(0, max_car_num + 1):  # 当天还车数量ret
            p_ret = poisson(ret, return_mean[loc])  # 还ret辆车的概率
            if p_ret < accurate:  # 精度限制
                continue
            s_next = min(s - rent + ret, max_car_num)
            # 下一步状态：租车+还车后的租车点汽车数量
            Tp[loc, s, s_next] += p_rent * p_ret  # 状态转移概率
```

2. **策略评估**

   根据给定策略π，更新所有状态的价值，直至状态价值函数收敛（这里使用最大变化<0.1）。迭代策略评估中，对每个状态采用相同的操作：根据给定策略，得到所有可能的单步转移后的即时收益和每个后继状态的旧的价值函数，利用这二者的期望来更新状态的价值函数。

$$v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s]$$

```python
# 进行策略评估
while True:
    old_value = value.copy()
    # 遍历所有状态
    for i in range(max_car_num + 1):
        for j in range(max_car_num + 1):
            new_state_value = value_update([i, j], policy[i, j], value)
            value[i, j] = new_state_value
    max_value_change = abs(old_value - value).max()
    print(f'max value change: {max_value_change}')
    if max_value_change < 0.1:
         break
```

```python
def value_update(state, action, last_value):
    """
    更新当前状态的价值函数
    :param state: [i,j] i代表第一个租赁点的汽车数量，j代表第二个租赁点的汽车数量
    :param action: 动作
    :param last_value: 上一个价值函数
    :return: 当前状态的价值函数
    """
    # 移车之后状态从state变成new_state
    temp_v = -np.abs(action) * move_cost  # 移车代价
    for m in range(0, max_car_num + 1):
        for n in range(0, max_car_num + 1):  # 对所有后继状态
            # temp_V 即是所求期望
            # Tp[0,i,j]表示第一个租赁点状态从i到j的概率
            # Tp[1,i,j]表示第二个租赁点状态从i到j的概率
            temp_v += Tp[0, new_state[0], m] * Tp[1, new_state[1], n] * (
                    reward[0, new_state[0]] + reward[1, new_state[1]] +
discount * last_value[m, n])
    return temp_v
```

3. **策略改善**

在当前策略基础上，贪婪地选取行为，使得后继状态价值增加最多；

具体方法为在当前状态下，遍历动作空间，分别计算出每个动作的价值函数，最大的价值函数，即是贪婪的要选取的策略。

```python
# 策略改进
# 当前状态[i,j]
old_action = policy[i, j]
action_value = []
# 遍历动作空间
for action in actions:
    if -j <= action <= i:  # valid action
          action_value.append(value_update([i, j], action, value))
    else:
          action_value.append(-np.inf)
```

```
action_value = np.array(action_value)
# 贪婪选择，选择价值函数最大的动作
new_action = actions[np.wheron_value == action_value.max())[0]]
policy[i, j] = np.random.choice(new_action)
```

4. **策略迭代**

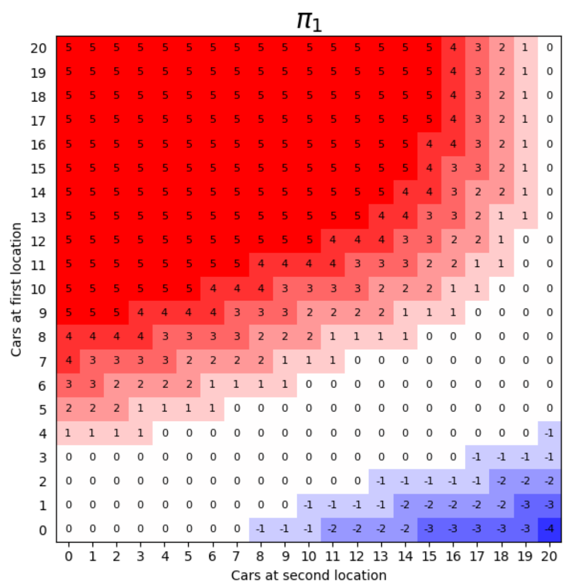
   在当前策略上迭代计算v值，再根据v值贪婪地更新策略，如此反复多次，最终得到最优策略和最优状态价值函数V

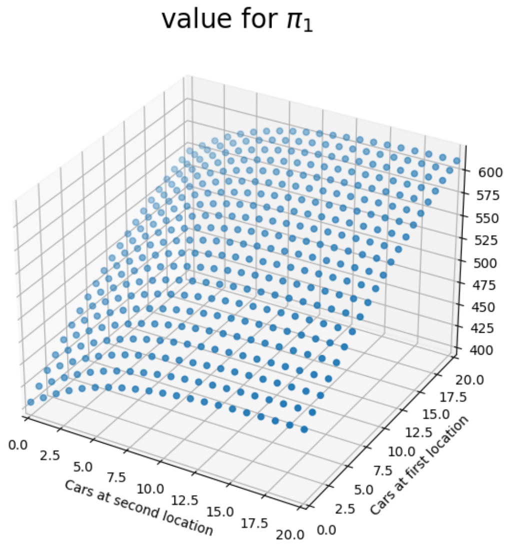
   对于本程序，迭代的停止条件为，对于当前策略，经过贪婪选择后，与旧策略相同即停止。

```
if policy_stable and (old_action not in new_action):
    policy_stable = False
```

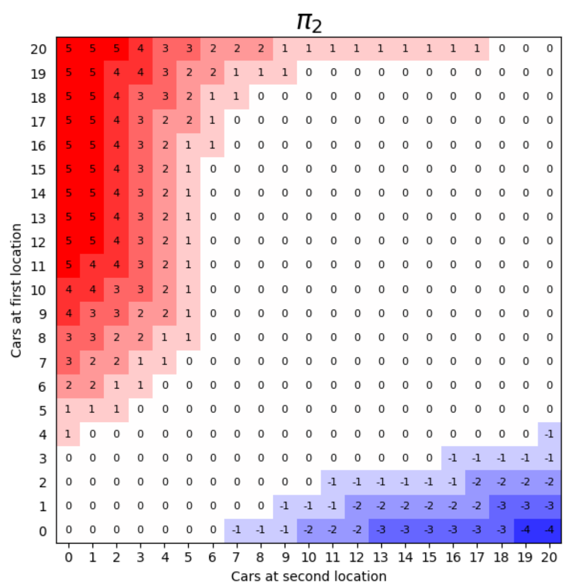
五、运行结果

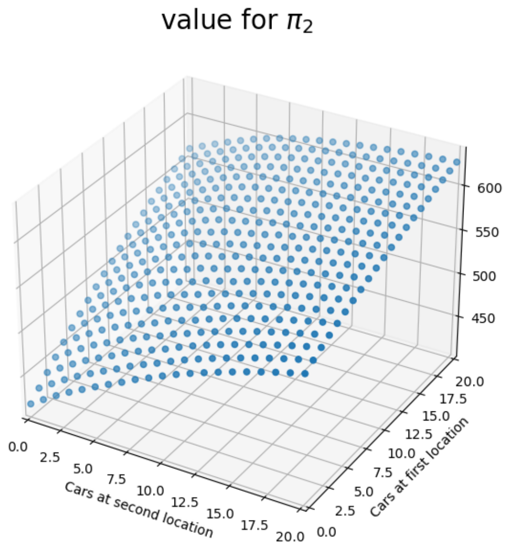
第一次策略迭代：



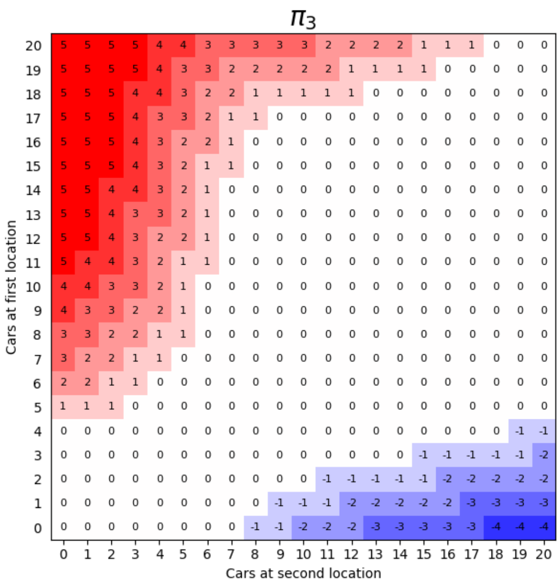


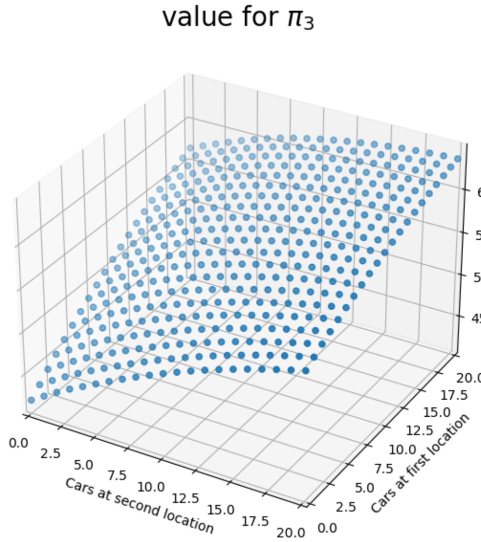
第二次策略迭代：



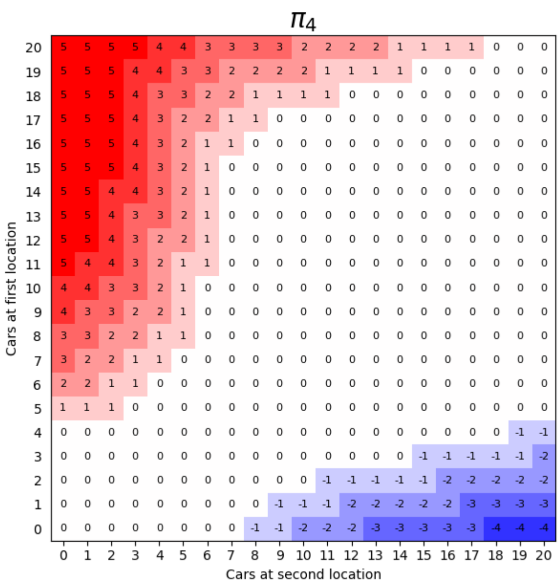


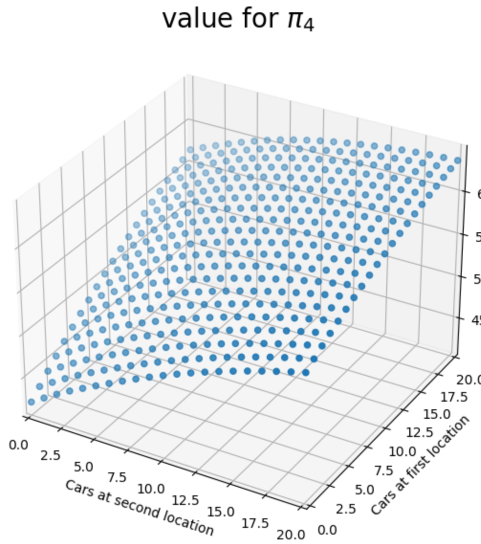
第三次策略迭代:





第四次策略迭代:





第五次策略迭代: (可以发现第五次跟第四次策略相同，所以停止迭代)