A8：Reinforcement Learning

刘帅 2020212267

1 摘要

1.1 问题描述

Jack有两个租车点，1号租车点和2号租车点，每个租车点最多可以停放20辆车。Jack每租出去一辆车可以获利10美金。每天租出去的车与收回的车的数量服从泊松分布λ^n/n! e^(−λ) 。每天夜里，Jack可以在两个租车点间进行车辆调配，每晚最多调配5辆车，且每调配一辆车花费2美金。

假设1号租车点租出去和收回的车辆服从λ=3的泊松分布，2号租车点租出去和收回的车辆数分别服从λ=4和λ=2的泊松分布。假设阻尼系数γ=0.09。

1.2 策略及分析

本次实验首先将分析上述问题对应的MDP四元组，并首先采用策略迭代的方法，求解最优调配策略。在此基础上，将进一步分析基于价值迭代的动态规划策略。策略求解简述如下：

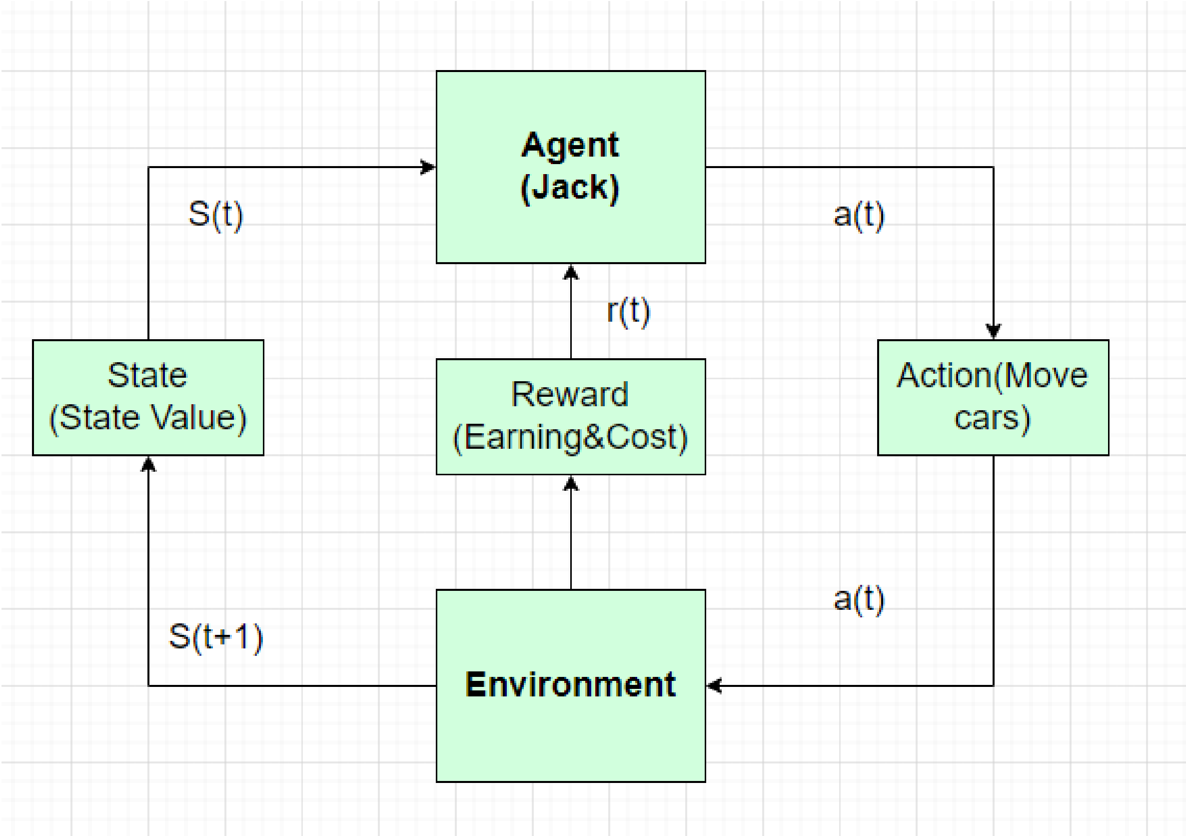
① 基于策略迭代的动态规划策略：

该策略共分为三个步骤。第一步，对状态价值和策略进行随机初始化。第二步，根据现有策略获取状态价值的估计值；第三步，根据期望最大化原则进行策略改进。迭代进行二、三步骤，直至状态价值和策略收敛于最优状态价值和最优策略。

② 基于价值迭代的动态规划策略：

价值迭代的动态规划策略: 该策略分为两个步骤。第一步，对状态价值和策略进行随机初始化。第二步，根据期望最大化原则，更新状态价值。重复第二步，直到状态价值收敛于最优状态价值。最后，根据最优状态价值确定最优策略。

③ 问题分析



状态 (State)： t 时刻下，Jack执行策略$\pi(a|s)$的两个租车地点持有汽车数目对应的状态价值，记作$v_\pi(s) = State_t(num_0, num_1)$，其中$num0, num1$表示两租车地点的车辆数目。

环境(Environment): Jack租车问题中的环境包括客户，负责与Jack进行交互。

智能体（Agent）： Jack是本问题中的智能体，负责根据环境的交互，作出使得自身期望收益最大化的动作。

动作（Action）： Jack可以选择在不同租车地点间进行车辆调配，同时，有移除车辆不多余租车地点持有车辆，两个租车点移动车辆数目相等的约束。

状态转移（Transition）： Jack做出移车动作后，环境接受动作，决定下一个状态。在这个问题中，认为杰克移车之后，用户同时进行租车与还车任务。

奖励(Reward)： 杰克每租出去一辆车可以获得奖励 10 美元，移动一辆车付出代价 2 美元

2 原理及推导

2.1 动态规划

在给定马尔可夫决策过程描述的环境模型下，每个MDP问题总有至少一个策略优于或等于其他所有策略，可利用动态规划的方法计算最优策略。强化学习问题中，我们利用贝尔曼最优方程求解，从而找到最优值函数和最优策略。

对于状态价值函数$V_\pi(s)$和状态-动作价值函数$q_\pi(s,a)$，有：

$v_\pi(s) = \mathbb{E}_\pi[q_\pi(s,a)] = \sum_a \pi(a|s) q_\pi(s,a)$

$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a)]$

$$\begin{aligned} v_*(s) &= max q_{\pi_*}(s,a) \\ &= \underset{a}{max} \mathbb{E}\pi_\theta[G_t | S_t = s, A_t = a] \\ &= \underset{a}{max} \mathbb{E}\pi_\theta[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \end{aligned}$$

$$= \underset{a}{max}\mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$$

$$= \underset{a}{max}\sum_{s'}\sum_{r}p(s',r|s,a)(r+\gamma v_*(s'))$$

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a)]$$

$$= \mathbb{E}[R_{t+1} + \gamma \underset{a'}{max} q_*(S_{t+1},a')|S_t = s, A_t = a]$$

$$= \sum_{s'}\sum_{r}p(s',r|s,a)(r+\gamma \underset{a'}{max} q_*(s',a'))$$

通过贝尔曼迭代方程可递归地更新公式，从而利用DP算法进行最优化，求解DP问题有基于策略迭代的DP策略和基于价值迭代的DP策略，下面将针对两种不同策略进行求解。

3 策略迭代

策略迭代的动态规划策略求解伪代码如下

$$\text{输入：} Environment, Agent$$
$$\text{输出：}\$v_*, \pi_*\$$$
$$1.\text{初始化}$$
$$\text{对} s \in S, \text{令} V(s) \in \mathbb{R} \text{以及} \pi(s) \in A(s)$$

$$2.\text{策略评估}$$
$$loop: \Delta \leftarrow 0$$
$$\text{对于每个循环，更新}$$
$$v \leftarrow V(s)$$
$$V(s) \leftarrow \sum_{s',r}p(s',r|s,a)(r+\gamma V(s'))$$
$$\Delta \leftarrow max\text{（}\Delta, |v - V(s)|) \text{直至} \Delta < \theta$$

$$3.\text{策略改进}$$
$$policy\_stable \leftarrow True \text{对于每一个循环}, \text{更新}$$
$$old\_action \leftarrow \pi(s)$$
$$\pi(s) \leftarrow \underset{a}{argmax}\sum_{s',r}p(s'r|s,a)(r+\gamma V(s'))$$
$$if\ old\_action \neq \pi(s):$$
$$policy\_stable \leftarrow False$$
$$if\ old\_action = \pi(s):$$
$$\text{迭代完成}$$

3.1 价值函数可解性证明

由于每个MDP问题总有至少一个策略优于或等于其他所有策略，对于任意一个策略π，有:

$$v_t(s) = E_\pi(G_t|S_t = s)$$
$$= E_\pi(R\_t + \gamma G_{t+1}|S_t = s)$$
$$= E_\pi(R_t + \gamma v_\pi(S_{t+1}|S_t = s))$$
$$= \sum_{a}\pi(a|s)\sum_{s'r}p(s',r|s,a)(r+\gamma v_\pi(s'))$$

只要$\gamma < 1$或者任何状态在π下都能保证中止，那么状态价值函数V_π一定唯一存在。且如果MDP四元组可知，那么上述公式为线性方程组，证明该价值函数具有可解性。

3.2 收敛性证明

利用贝尔曼方程求近似解，有：

$$\begin{aligned} v_{k+1}(s) &= E_\pi(R_{t+1} + \gamma v_k(S_{t+1})|S_t = s) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)(r + \gamma v_k(s')) \end{aligned}$$

因此，当k趋于无穷的时候v_k会收敛到v_π。

3.3 策略迭代

对于已经确定的策略π，我们已经确定了它的价值函数v_π。但无法判断我们目前的策略是否最优，因此，我们可以延伸到所有状态的所有可能动作，在每个状态下根据动作-价值函数$q_\pi(s,a)$选择最优的。有：

$$\begin{aligned} \pi'(s) &= \underset{a}{argmax} E(R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a) \\ &= \underset{a}{argmax} \sum_{s',r} p(s',r|s,a)(r + \gamma v_\pi(s')) \end{aligned}$$

因此只要该策略生成了更好的策略，我们就可以不断进行改进和迭代，并求出该状态的价值函数。针对Jack租车问题，则是要求任意状态时，如何对11个调配动作（挪动范围从-5到5）进行安排和优化。

3.4 代码部分

3.4.1 常量定义

```
from matplotlib import pyplot as plt
import numpy as np
from scipy.stats import poisson
import seaborn as sns

parkingnum = 20 #每个租车点最多可以停放20辆车
lamda1_rent = 3 #停车场1租车λ值（3）
lamda1_return = 3 #停车场1还车λ值（3）
lamda2_rent = 4 #停车场2租车λ值（4）
lamda2_return = 2 #停车场2还车λ值（2）
MAX_ACTION = 5 #最大调配汽车数目（5）
DISCOUNT = 0.09 #收益折扣
cost = 2 #每调配一辆车花费2美金。
earn = 10 #租车的收入（10）
actions = np.arange(-MAX_ACTION, MAX_ACTION + 1) #动作集合（-5，-4，…，4，5）
poisson_max = 11 #限制泊松分布产生请求数目的上限
poisson_cache = dict() # 存储每个（n，λ）对应的泊松概率，key为n*(poisson_max-1)+lam
```

3.4.2 泊松分布

储存每个（n，λ）对应的泊松概率，pmf记录泊松分布的离散值

```
def poisson_prob(n, lam):
    global poisson_cache
    key = n * (poisson_max - 1) + lam
    if key not in poisson_cache:
        poisson_cache[key] = poisson.pmf(n, lam)
    return poisson_cache[key]
```

3.4.3 dp算法

```
class dp:
    def __init__(self):
        self.v = np.ones((parkingnum + 1, parkingnum + 1), float)
        self.actions = np.zeros((parkingnum + 1, parkingnum + 1), int)
        self.gamma = DISCOUNT
        self.delta = 0
        self.theta = 0.01
        pass

    def state_value(self, state, action, state_value, constant_returned_cars):
        """
        :state: 状态定义为每个地点的车辆数
        :action: 车辆的移动数量[-5,5]，共11个动作
        :state_value: 状态价值矩阵
        :constant_returned_cars: 将换车的数目设定为泊松均值，替换为泊松概率分布
        """
        returns = 0.0
        # 移动车辆产生负收益
        returns -= cost * abs(action)
        # 移动后的车辆总数不能超过20
        NUM_OF_CARS_1 = min(state[0] - action, parkingnum)
        NUM_OF_CARS_2 = min(state[1] + action, parkingnum)
        # 遍历两地全部的可能概率下租车请求数目
        for rent_1 in range(poisson_max):
            for rent_2 in range(poisson_max):
                # prob为两地租车请求的联合概率，概率为泊松分布
                prob = poisson_prob(rent_1, lamda1_rent) * poisson_prob(rent_2,
lamda2_rent)
                # 两地原本汽车数量
                num_of_cars_1 = NUM_OF_CARS_1
                num_of_cars_2 = NUM_OF_CARS_2
                # 有效租车数目必须小于等于该地原有的车辆数目
                valid_rent_1 = min(num_of_cars_1, rent_1)
                valid_rent_2 = min(num_of_cars_2, rent_2)
                # 计算回报，更新两地车辆数目变动
                reward = (valid_rent_1 + valid_rent_2) * earn
                num_of_cars_1 -= valid_rent_1
                num_of_cars_2 -= valid_rent_2
                # 如果还车数目为泊松分布的均值
                if constant_returned_cars:
                    # 两地的还车数目均为泊松分布均值
                    returned_cars_1 = lamda1_return
                    returned_cars_2 = lamda2_return
                    # 还车后总数不能超过车场容量
                    num_of_cars_first_loc = min(num_of_cars_1 + returned_cars_1,
parkingnum)
```

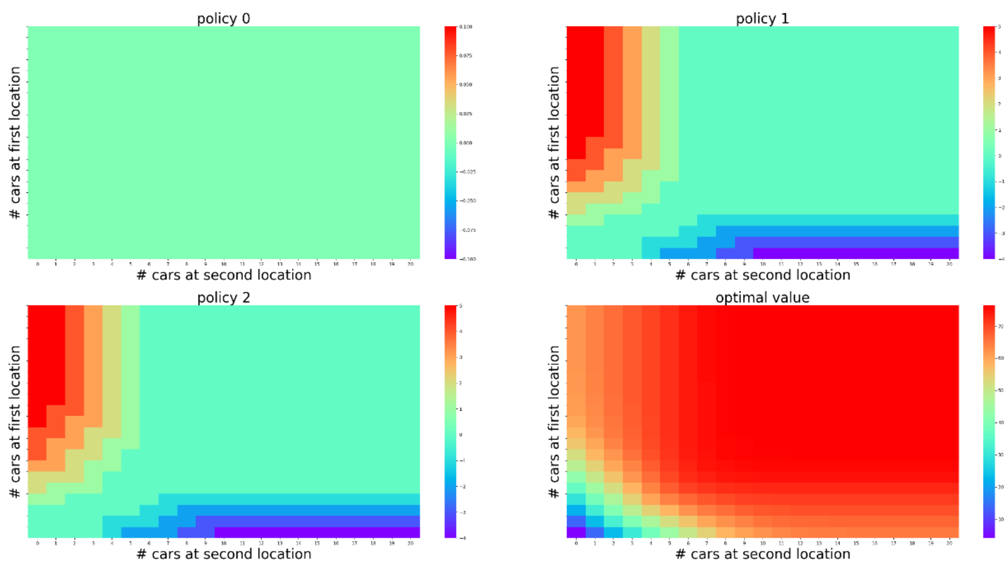
```
                    num_of_cars_second_loc = min(num_of_cars_2 +
returned_cars_2, parkingnum)
                    # 核心：
                    # 策略评估：V(s) = p(s',r|s,π(s))[r + γV(s')]
                    returns += prob * (reward + DISCOUNT *
state_value[num_of_cars_first_loc, num_of_cars_second_loc])
        return returns
```

3.4.4 策略迭代

```
    def policy_iteration(self):
        # 设置迭代参数
        iterations = 0
        # 准备画布大小，并准备多个子图
        _, axes = plt.subplots(2, 3, figsize=(40, 20))
        # 调整子图的间距，wspace=0.1为水平间距，hspace=0.2为垂直间距
        plt.subplots_adjust(wspace=0.1, hspace=0.2)
        # 这里将子图形成一个1*6的列表
        axes = axes.flatten()
        while True:
            # 使用seaborn的heatmap作图
            fig = sns.heatmap(np.flipud(self.actions), cmap="rainbow",
ax=axes[iterations])
            # 定义标签与标题
            fig.set_ylabel('# cars at first location', fontsize=30)
            fig.set_yticks(list(reversed(range(parkingnum + 1))))
            fig.set_xlabel('# cars at second location', fontsize=30)
            fig.set_title('policy {}'.format(iterations), fontsize=30)
            # policy evaluation (in-place) 策略评估（in-place）
            # 未改进前，第一轮policy全为0，即[0，0，0...]
            while True:
                old_value = self.v.copy()
                for i in range(parkingnum + 1):
                    for j in range(parkingnum + 1):
                        # 更新V（s）
                        new_state_value = self.state_value([i, j],
self.actions[i, j], self.v)
                        self.v[i, j] = new_state_value
                # 比较V_old(s)、V(s)，收敛后退出循环
                max_value_change = abs(old_value - self.v).max()
                print('max value change {}'.format(max_value_change))
                if max_value_change < 1e-4:
                    break

            # policy improvement
            #收敛到实际最优策略。
            policy_stable = True
            # i、j分别为两地现有车辆总数
            for i in range(parkingnum + 1):
                for j in range(parkingnum + 1):
                    old_action = self.actions[i, j]
                    action_returns = []
                    # actions为全部的动作空间，即[-5、-4...4、5]
                    for action in actions:
                        if (0 <= action <= i) or (-j <= action <= 0):
```

```
              action_returns.append(self.state_value([i, j],
action, self.v))
            else:
              action_returns.append(-np.inf)
          # 找出产生最大动作价值的动作
          new_action = actions[np.argmax(action_returns)]
          # 更新策略
          self.actions[i, j] = new_action
          if policy_stable and old_action != new_action:
            policy_stable = False
      print('policy stable {}'.format(policy_stable))

      if policy_stable:
        fig = sns.heatmap(np.flipud(self.v), cmap="rainbow",
ax=axes[-1])
        fig.set_ylabel('# cars at first location', fontsize=30)
        fig.set_yticks(list(reversed(range(parkingnum + 1))))
        fig.set_xlabel('# cars at second location', fontsize=30)
        fig.set_title('optimal value', fontsize=30)
        break

      iterations += 1

    plt.savefig('./policy_iteration.png')
    plt.show()
    plt.close()
    return
```

3.5 实验结果




```
max value change 74.19309348518719
max value change 5.8677560301762455
max value change 0.35842087965759983
max value change 0.027912986621O4147
max value change 0.0019710407457615986
max value change 0.00013182236501130262
max value change 8.519945697393894e-06
```

```
policy stable False
max value change 27.10825516178984
max value change 0.19225821383447794
max value change 0.010635626186877 19
max value change 0.0008548378402570833
max value change 5.242054288601139e-05
policy stable False
max value change 0.13192952374075873
max value change 0.002041987470249751
max value change 7.774259248094495e-05
policy stable True
```

随着策略不断迭代，策略逐渐收敛到最优策略，同时状态价值函数也逐渐收敛。由图可知，当一个租车点汽车少，另一个汽车点汽车多是，则会将更多的汽车从汽车多的租车地点转移到汽车少的租车地点，且高价值状态更加靠近车数较多的租车点上。

4 价值迭代

由于策略迭代算法在每一次更换策略后都需要进行策略评估，因此迭代过程所需要的计算量和耗时都显著变长。因此，我们可以在遍历一次以后立刻停止策略评估，用公式表达为：

$$\begin{aligned} v_{k+1}(s) &= \underset{a}{max} E(R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a) \\ &= \underset{a}{max} \sum_{s',r} p(s', r | s, a)(r + \gamma v_k(s')) \end{aligned}$$

基于价值迭代的动态规划伪代码如下：

$$\begin{gathered} loop : \Delta \leftarrow 0 \\ \text{对于每个循环，更新} \\ v \leftarrow V(s) \\ V(s) \leftarrow = \underset{a}{max} \sum_{s',r} p(s', r | s, a)(r + \gamma v_k(s')) \\ \Delta \leftarrow max(\Delta, |v - V(s)|) \\ \pi(s) = \underset{a}{argmax} \sum_{s',r} p(s', r | s, a)(r + \gamma v_\pi(s')) \end{gathered}$$

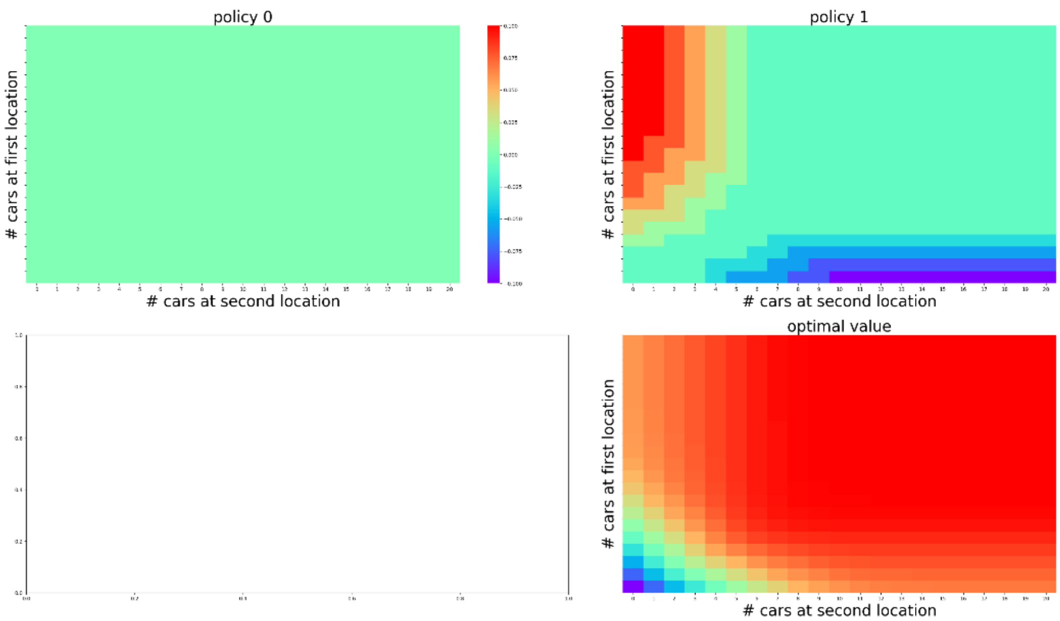
4.1代码部分

```
    def value_iteration(self):
        # 设置迭代参数
        iterations = 0

        # 准备画布大小，并准备多个子图
        _, axes = plt.subplots(2, 2, figsize=(40, 20))
        # 调整子图的间距，wspace=0.1为水平间距，hspace=0.2为垂直间距
        plt.subplots_adjust(wspace=0.1, hspace=0.2)
        # 这里将子图形成一个1*4的列表
        axes = axes.flatten()
        while True:
            # 使用seaborn的heatmap作图
            fig = sns.heatmap(np.flipud(self.actions), cmap="rainbow",
ax=axes[iterations])

            # 定义标签与标题
```

```python
            fig.set_ylabel('# cars at first location', fontsize=30)
            fig.set_yticks(list(reversed(range(parkingnum + 1))))
            fig.set_xlabel('# cars at second location', fontsize=30)
            fig.set_title('policy {}'.format(iterations), fontsize=30)

            # value iteration 价值迭代
            while True:
                old_value = self.v.copy()
                for i in range(parkingnum + 1):
                    for j in range(parkingnum + 1):
                        action_returns = []
                        # actions为全部的动作空间，即[-5、-4...4、5]
                        for action in actions:
                            if (0 <= action <= i) or (-j <= action <= 0):
                                action_returns.append(self.state_value([i, j], action, self.v))
                            else:
                                action_returns.append(-np.inf)
                        # 找出产生最大动作价值的动作
                        max_action = actions[np.argmax(action_returns)]
                        # 更新V（s）
                        new_state_value = self.state_value([i, j], max_action, self.v)
                        # in-place操作
                        self.v[i, j] = new_state_value
                # 比较V_old(s)、V(s)，收敛后退出循环
                max_value_change = abs(old_value - self.v).max()
                print('max value change {}'.format(max_value_change))
                if max_value_change < 1e-4:
                    break

            # policy improvement
            policy_stable = True
            # i、j分别为两地现有车辆总数
            for i in range(parkingnum + 1):
                for j in range(parkingnum + 1):
                    old_action = self.actions[i, j]
                    action_returns = []
                    # actions为全部的动作空间，即[-5、-4...4、5]
                    for action in actions:
                        if (0 <= action <= i) or (-j <= action <= 0):
                            action_returns.append(self.state_value([i, j], action, self.v))
                        else:
                            action_returns.append(-np.inf)
                    # 找出产生最大动作价值的动作
                    new_action = actions[np.argmax(action_returns)]
                    # 更新策略
                    self.actions[i, j] = new_action
                    if policy_stable and old_action != new_action:
                        policy_stable = False
            print('policy stable {}'.format(policy_stable))

            if policy_stable:
```

```
                fig = sns.heatmap(np.flipud(self.v), cmap="rainbow", ax=axes[-1])
                fig.set_ylabel('# cars at first location', fontsize=30)
                fig.set_yticks(list(reversed(range(parkingnum + 1))))
                fig.set_xlabel('# cars at second location', fontsize=30)
                fig.set_title('optimal value', fontsize=30)
                break

            iterations += 1

        plt.savefig('./value_iteration.png')
        plt.show()
        plt.close()
        return
```

4.2 实验结果



随着状态价值的不断迭代策略逐渐收敛到最优策略，状态价值也逐渐收敛。在价值迭代算法中，jack学习到的规则为:在一个租车地点的汽车少,而另一个租车地点的汽车多时，会将更多的汽车从汽车多的租车地点转移到汽车少的租车地点。该方法通过不需要先遍历所有状态进行估计，而是边估计边改进的算法，更加接近DP算法的本智，该算法更为高效和快速。