第3章 蒙特卡罗法

CHAPTER 3

动态规划法的求解过程需要完整明确的环境模型信息，即环境模型的状态转移概率，在这种环境模型上进行的强化学习方法叫作基于模型的方法(Model-Based Method)，但许多环境模型的状态转移概率是未知的或不完整的，在这种环境模型上进行的强化学习方法叫作免模型法(Model-Free Method)。本章将要介绍的蒙特卡罗法(Monte Carlo Method)就是一种免模型强化学习法。

蒙特卡罗法和动态规划法的相同之处在于也使用了策略迭代的算法框架，策略评估和策略改进交替进行，直到获得最优值函数和最优策略；不同之处在于策略评估的过程，动态规划法通过求解贝尔曼方程组来评估策略，而蒙特卡罗法通过采样得到大量经验轨迹(Experience)并计算经验轨迹的回报的算术均值来获得动作值的估计，从而实现策略评估。为了保证采样过程能获得较正常的回报数据，本章的讨论限制在离散型强化学习范围，这意味着环境模型的状态和动作空间都是有限离散的，而且能在有限时间步达到终止状态。

本章首先简单介绍蒙特卡罗法，然后重点讨论蒙特卡罗策略评估和策略改进，接着讨论同策略和异策略蒙特卡罗强化学习算法，最后介绍处理大规模离散状态空间强化学习任务的蒙特卡罗树搜索。

14min

3.1 蒙特卡罗法简介

蒙特卡罗法，也称统计模拟方法，是20世纪40年代中期被提出的一种以概率统计理论为指导的数值计算方法。蒙特卡罗法的核心思想是使用随机数(或更常见的伪随机数)来大量重复地随机采样，通过对采样结果进行统计分析而得到数值结果。根据大数定理，蒙特卡罗法采样越多，通过对采样进行分析而得到的统计结果就越接近真实值。蒙特卡罗法在金融工程学、宏观经济学、计算物理学(如粒子输运计算、量子热力学计算、空气动力学计算)等领域应用广泛。

蒙特卡罗法的基本步骤如下。

步骤1：构造或描述概率过程，构造一个概率过程，它的某些参量正好是所要求的问题

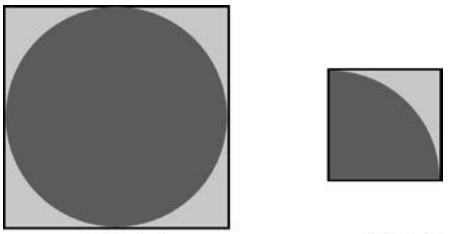
的解。这一步通常要将不具有随机性质的问题转换为具有随机性质的问题。

步骤 2：实现从已知概率分布抽样，根据已知的概率分布进行大量的随机抽样。

步骤 3：对随机抽样结果进行统计分析，确定一些无偏估计量，它们对应着所要求的问题的解。利用抽样结果对这些无偏估计量进行估计，估计结果就是解的近似。

以下来看一个用蒙特卡罗法求解问题的例子。

【例 3-1】　用蒙特卡罗法估计 π 值。设有一半径为 r 的圆及其外切正方形，如图 3-1(a)所示。



(a) 理论方案　　(b) 实际方案

图 3-1　蒙特卡罗法估计 π 值示意图

向该正方形随机地投掷 n 个点，设落入圆内的点数为 k 个，由于所投入的点在正方形上均匀分布，因而所投入的点落入圆内的概率为

$$\frac{\pi r^2}{4r^2}=\frac{\pi}{4} \tag{3-1}$$

所以，当 n 足够大时，k 与 n 之比就逼近这一概率，即 $\pi/4$，从而 $\pi\approx 4k/n$。在具体实现时，只要在第一象限计算就行，如图 3-1(b)所示。用蒙特卡罗法估计 π 值的 Python 代码如下：

```
##【代码 3-1】蒙特卡罗法估计 π 值

import random
import math

random.seed(0)                                  #初始化随机数种子
n = 1000000                                     #投点数
k = 0                                           #落在圆内的点计数
for _ in range(n):
    x1,x2 = random.random(),random.random()     #随机生成一个点
    if math.sqrt(x1 * x1 + x2 * x2)<=1:         #如果点落在圆内
        k += 1
print("pi = ",4 * k/n)
```

运行结果如下：

```
pi = 3.14244
```

3.2 蒙特卡罗策略评估

动态规划法是基于模型的方法，需要完整明确的模型信息，也就是环境模型的状态转移概率，这种模型也称为概率模型，但现实中的大部分强化学习任务不能直接得到准确的概率模型，解决这一矛盾有以下两个方案：

(1) 运用机器学习算法学习环境模型，即学习一种状态转移概率函数。当然，这需要大量的真实数据作为训练数据。此处“真实”是指数据要通过物理方式获得，而不是通过计算机模拟方式获得。对于实验成本较低的环境，大量真实数据是容易获得的，但对于实验成本较高的环境，这个方案就不适用了。

(2) 免模型强化学习。许多环境虽然得不到具有准确状态转移概率的概率模型，但可以根据物理机理得到一个可以进行随机抽样的模型，这种模型称为抽样模型。免模型强化学习就是运行在抽样模型上的强化学习。本章要讨论的蒙特卡罗强化学习法就是一种免模型强化学习。

3.2.1 蒙特卡罗策略评估

与策略迭代算法(算法 2-3)一样，蒙特卡罗强化学习法也使用“策略评估—策略改进”交替进行的算法框架来设计，但因为环境的状态转移概率未知，所以不能通过解贝尔曼方程组的方式来计算值函数。本节介绍基于蒙特卡罗采样的策略评估方法。

设当前策略为 π，在 π 下一个经历完整的 MDP 序列是指从初始状态出发，使用 π 进行动作选择，按照抽样模型进行状态转移，一直到达终止状态的 MDP 序列，即

$$s_0,a_0,R_1,s_1,a_1,R_2,\cdots,s_t,a_t,R_{t+1},a_{t+1},\cdots,s_{T-1},a_{T-1},R_T,s_T \tag{3-2}$$

其中，s_T 是终止状态，R_T 假设由二元奖励函数得到，所以使用大写英文字母。

假设在策略 π 下经过大量局数(m 个 Episode)的随机环境交互得到了 m 条经历完整的 MDP 序列，即

$$\begin{aligned}
&s_0^{<1>},a_0^{<1>},R_1^{<1>},s_1^{<1>},a_1^{<1>},R_2^{<1>},\cdots,s_{T_m-1}^{<1>},a_{T_m-1}^{<1>},R_{T_m}^{<1>},s_{T_m}^{<1>}\\
&s_0^{<2>},a_0^{<2>},R_1^{<2>},s_1^{<2>},a_1^{<2>},R_2^{<2>},\cdots,s_{T_m-1}^{<2>},a_{T_m-1}^{<2>},R_{T_m}^{<2>},s_{T_m}^{<2>}\\
&\vdots\\
&s_0^{<m>},a_0^{<m>},R_1^{<m>},s_1^{<m>},a_1^{<m>},R_2^{<m>},\cdots,s_{T_m-1}^{<m>},a_{T_m-1}^{<m>},R_{T_m}^{<m>},s_{T_m}^{<m>}
\end{aligned} \tag{3-3}$$

这里 $s_{T_i}^{<i>}$，$i=1,2,\cdots,m$ 均为终止状态，需要注意的是 $s_{T_1}^{<1>},\cdots,s_{T_m}^{<m>}$ 是一样的状态(假设只有一个终止状态)，但每局的交互达到终止状态需要的时间步可能是不一样的，即 $T_1,T_2,\cdots,T_m$ 可能是不同的。我们的目标是评估策略，即求策略 π 下的动作值

$$Q_\pi(s,a)=E\left[R_t+\gamma R_{t+1}+\cdots+\gamma^{T-t-1}R_T \mid S_t=s,A_t=a\right] \tag{3-4}$$

按照样本均值是期望的无偏估计这一原理，只需求出所有经历完整 MDP 序列中从出现状态-动作对(s,a)开始算起的累积折扣奖励的均值，并用均值来近似期望，因此，对任意

状态-动作对(s,a)，计算其首次出现在某一序列时的累积折扣奖励

$$G^{<i>}(s,a)=R_t^{<i>}+\gamma R_{t+1}^{<i>}+\cdots+\gamma^{T_i-t-1}R_{T_i}^{<i>}\mid s_t^{<i>}=s,a_t^{<i>}=a,\quad i=1,2,\cdots,m \tag{3-5}$$

当然(s,a)并不一定会在序列中出现，若不出现，则不计算累积折扣奖励。

设指示函数

$$I^{<i>}(s,a)=\begin{cases}1 & (s,a)\text{ 出现在第 }i\text{ 条序列中}\\ 0 & (s,a)\text{ 未出现在第 }i\text{ 条序列中}\end{cases} \tag{3-6}$$

则状态-动作对(s,a)的动作值样本均值为

$$\bar{Q}_\pi(s,a)=\frac{\sum_{i=1}^{m}G^{<i>}(s,a)}{\sum_{i=1}^{m}I^{<i>}(s,a)} \tag{3-7}$$

值得说明的是，许多资料在介绍蒙特卡罗策略评估时计算的是状态值均值，但蒙特卡罗强化学习算法需要的是动作值。在动态规划中，这并没有问题，因为根据已知的状态转移概率可以从状态值计算动作值，但蒙特卡罗强化学习的基本假设是状态转移概率未知，所以不能从状态值计算动作值，所以本书在介绍蒙特卡罗策略评估时直接计算动作值均值。

用式(3-5)计算累积折扣奖励时，是从首次访问到序列中出现该状态-动作对时开始的，但同一种状态-动作对可能在一个经历完整的 MDP 序列中多次出现。此时，也可以在每次访问序列中出现该状态-动作对时都计算一次累积折扣奖励。前一种计算范式称为首次访问(First Visit)，后一种计算范式称为每次访问(Every Visit)。显然在 MDP 样本序列一定的情况下，每次访问比首次访问计算量更大，但在 MDP 样本序列较少的情况下，使用每次访问能获取更多的信息。本书主要探讨首次访问范式。

首次访问蒙特卡罗策略评估算法的流程可以总结如下：

算法 3-1　首次访问蒙特卡罗策略评估算法

1. 输入：环境模型 MDP$(\mathbf{S},\mathbf{A},R,\gamma)$，待评估的策略 π，要生成的 MDP 序列条数 m
2. 初始化：累积回报 $G_{\text{sum}}(s,a)=0$，计数器 $K(s,a)=0$
3. 过程：随机抽样，根据当前策略 π，抽样产生 m 条经历完整的 MDP 序列
4. 　　循环：$i=1\sim m$
5. 　　循环：依次遍历第 i 条 MDP 序列中的所有状态-动作对(s,a)
6. 　　若(s,a)首次出现在 MDP 序列中，则根据式(3-5)计算累积折扣奖励
7. 　　累积回报：$G_{\text{sum}}(s,a)\leftarrow G_{\text{sum}}(s,a)+G^{<i>}(s,a)$
8. 　　更新计数：$K(s,a)\leftarrow K(s,a)+1$
9. 　　策略评估：$\bar{Q}_\pi(s,a)\leftarrow G_{\text{sum}}(s,a)/K(s,a)$
10. 输出：样本均值 $\bar{Q}_\pi$ 作为动作值的近似

3.2.2 增量式蒙特卡罗策略评估

算法 3-1 是在所有的 MDP 样本序列都已抽样完成后才计算累积折扣奖励和样本均值的，这样做的效率非常低。其实，可以使用计算样本均值的增量法，每新增一条 MDP 序列就更新一次样本均值。先看计算样本均值的增量法。

$$\begin{aligned}\bar{x}_k &= \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) = \frac{1}{k}\left[x_k + (k-1)\bar{x}_{k-1}\right] \\ &= \bar{x}_{k-1} + \frac{1}{k}(x_k - \bar{x}_{k-1})\end{aligned} \tag{3-8}$$

式(3-8)表明，要计算序列$\{x_1, x_2, \cdots, x_k\}$的均值$\bar{x}_k$，只需知道序列$\{x_1, x_2, \cdots, x_{k-1}\}$的均值$\bar{x}_{k-1}$和项$x_k$。将这一原理用于计算动作值可得

$$\bar{Q}_\pi^{<k>}(s,a) = \bar{Q}_\pi^{<k-1>}(s,a) + \frac{1}{k}(G^{<i>}(s,a) - \bar{Q}_\pi^{<k-1>}(s,a)) \tag{3-9}$$

这里计数器k是指状态-动作对(s,a)在第i条序列中首次出现时已经在之前的各序列中首次出现过的总次数，包括在第i条序列中出现的这一次。

增量式首次访问蒙特卡罗策略评估算法的流程可以总结如下：

算法 3-2 增量式首次访问蒙特卡罗策略评估算法

1. 输入：环境模型 MDP$(\mathbf{S}, \mathbf{A}, R, \gamma)$，待评估的策略$\pi$，要生成的 MDP 序列条数$m$
2. 初始化：动作值$\bar{Q}_\pi(s,a)=0$，计数器$K(s,a)=0$
3. 过程：
4. 　　循环：$i=1\sim m$
5. 　　随机抽样：根据当前策略π，抽样产生一条经历完整的 MDP 序列
6. 　　循环：依次遍历序列中的所有状态-动作对(s,a)
7. 　　若(s,a)首次出现在 MDP 序列中，则根据式(3-5)计算累积折扣奖励
8. 　　更新计数：$K(s,a) \leftarrow K(s,a)+1$
9. 　　策略评估：根据式(3-9)计算样本均值
10. 输出：样本均值$\bar{Q}_\pi$作为动作值的近似

相较于首次访问蒙特卡罗策略评估算法，增量式首次访问蒙特卡罗策略评估算法更加灵活高效。算法 3-2 中，循环次数m是可变的，可以根据策略改进的结果动态地调整m的值；另外，可以将策略改进的过程直接嵌入策略评估过程中，以提升整个算法的效率，这一点将在 3.3 节中详细讨论，接下来通过两个例子进一步理解蒙特卡罗策略评估。

3.2.3 蒙特卡罗策略评估案例

【例 3-2】 21 点游戏是赌场上一种常见的赌博游戏，也是影视剧中经常出现的桥段。21 点游戏使用一副或多副标准的 52 张纸牌，其中 2～10 的牌的点数按面值计算；J、Q、K

都算作 10 点；A 可算作 1 点，也可以算作 11 点，如果总点数不超过 21 点，则必须算作 11 点，此时 A 称为可用的(Usable)，否则算作 1 点。玩家的目标是所抽牌的总点数比庄家的牌更接近 21 点，但不超过 21 点。

首先，庄家以顺时针方向依次向众玩家和自己派发一张暗牌和一张明牌，然后庄家以顺时针方向逐位询问玩家是否需要再要牌(以明牌方式派发)，玩家可以选择继续要牌(Hits)，也可以选择放弃要牌(Sticks)，在要牌过程中，如果玩家所有牌加起来超过 21 点，玩家就输了(俗称爆煲，Bust)，游戏结束，该玩家的筹码归庄家。

如果玩家无爆煲，庄家询问完所有玩家后，就必须揭开自己手上的暗牌。若庄家总点数小于 17 点，就必须继续要牌；如果庄家爆煲，便向没有爆煲的玩家赔出该玩家所投的同等筹码。如果庄家无爆煲且大于或等于 17 点，则庄家与玩家比较点数决胜负，大的为赢，点数相同则为平手。

21 点游戏属于对战类强化学习问题。为了简单起见，假设只有一个玩家，即对战双方是庄家和玩家。在本例中，将玩家看作智能体，庄家和庄家的策略看作环境的一部分。玩家有两个可供选择的动作：Hits 和 Sticks，所以动作空间为

$$\mathbf{A}=\{\text{Hits},\text{Sticks}\}$$

玩家根据自己手中的总点数(Player)，庄家的明牌点数(Dealer)和自己手中是否有可用的 A(Ace，Ace=True 表示 A 算作 11 点，Ace=False 表示 A 算作 1 点)来决定是否继续要牌，所以状态空间为

$$\mathbf{S}=\{\text{Player},\text{Dealer},\text{Ace}\}$$

Gym 库中集成了 21 点游戏的模型环境 Blackjack-v0，可以直接调用。为了更加便于读者理解，本例对 Gym 中的集成模型进行了适当简化，代码如下：

```
##【代码 3-2】简化版 21 点游戏环境模型

import numpy as np
from gym import spaces
from gym.utils import seeding

#1 = Ace, 2-10 = Number cards, Jack/Queen/King = 10
deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]

#发牌函数
def draw_card(np_random):
    return int(np_random.choice(deck))

#首轮发牌函数
def draw_hand(np_random):
    return [draw_card(np_random), draw_card(np_random)]

#判断是否有可用 Ace
def usable_ace(hand):
```

```
    return 1 in hand and sum(hand) + 10 <= 21

#计算手中牌的总点数
def sum_hand(hand):
    if usable_ace(hand):
        return sum(hand) + 10
    return sum(hand)

#判断是否爆煲
def is_bust(hand):
    return sum_hand(hand) > 21

class BlackjackEnv():
    def __init__(self):
        self.action_space = spaces.Discrete(2)
        self.observation_space = spaces.Tuple((
            spaces.Discrete(32),
            spaces.Discrete(11),
            spaces.Discrete(2)))
        self.seed()
        self.reset()

    def seed(self, seed = None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def step(self, action):
        if action:                                      #叫牌
            self.player.append(draw_card(self.np_random))
            if is_bust(self.player):                    #如果爆煲
                done = True
                reward = -1.
                info = 'Player bust'
            else:                                       #如果没有爆煲,则继续
                done = False
                reward = 0.
                info = 'Keep going'
        else:                                           #停牌
            while sum_hand(self.dealer) < 17:           #如果庄家点数小于17,则继续叫牌
                self.dealer.append(draw_card(self.np_random))
            if is_bust(self.dealer):                    #如果爆煲
                reward = 1
                info = 'Dealer bust'
            else:
                reward = np.sign(
                        sum_hand(self.player) - sum_hand(self.dealer))
                if reward == 1: info = 'Player win'
```

```
                elif reward == 1: info = 'Drawing'
                else: info = 'Dealer win'
            done = True

        return self._get_obs(), reward, done, info

    def _get_obs(self):
        return (sum_hand(self.player),
                self.dealer[0], usable_ace(self.player))

    def reset(self):
        self.dealer = draw_hand(self.np_random)
        self.player = draw_hand(self.np_random)
        return self._get_obs()
```

将环境模型代码命名为 blackjack.py，并与本章后面相关例题的代码放在同一个文件夹中调用。

假设玩家的一个简单策略为手中牌的总点数大于或等于 18 点则停牌，否则继续叫牌。用蒙特卡罗首次访问策略评估算法计算该策略下的动作值的代码如下：

```
##【代码 3-3】用蒙特卡罗首次访问策略评估算法代码

import numpy as np
import blackjack
from collections import defaultdict

'''
待评估的玩家策略:如果点数小于 18,则继续叫牌,否则停牌
'''
def simple_policy(state):
    player, dealer, ace = state
    return 0 if player >= 18 else 1              #0:停牌,1:要牌

'''
首次访问蒙特卡罗策略评估:算法 3-1 的具体实现
'''
def firstvisit_mc_actionvalue(env,num_episodes=50000):
    r_sum = defaultdict(float)                   #记录状态-动作对的累积折扣奖励之和
    r_count = defaultdict(float)                 #记录状态-动作对的累积折扣奖励次数
    r_Q = defaultdict(float)                     #动作值样本均值

    #采样 num_episodes 条经验轨迹
    MDPsequence = []                             #经验轨迹容器
    for i in range(num_episodes):
```

```
            state = env.reset()                                    #环境状态初始化
            #采集一条经验轨迹
            onesequence = []
            while True:
                action = simple_policy(state)                      #根据给定的简单策略选择动作
                next_state,reward,done,_ = env.step(action)        #交互一步
                onesequence.append((state, action, reward))        #MDP 序列
                if done:                                           #游戏是否结束
                    break
                state = next_state
            MDPsequence.append(onesequence)

        #计算动作值,即策略评估
        for i in range(len(MDPsequence)):
            onesequence = MDPsequence[i]
            #计算累积折扣奖励
            SA_pairs = []
            for j in range(len(onesequence)):
                sa_pair = (onesequence[j][0],onesequence[j][1])
                if sa_pair not in SA_pairs:
                    SA_pairs.append(sa_pair)
                    G = sum([x[2] * np.power(env.gamma, k) for
                             k, x in enumerate(onesequence[j:])])
                    r_sum[sa_pair] += G                            #合并累积折扣奖励
                    r_count[sa_pair] += 1                          #记录次数
        for key in r_sum.keys():
            r_Q[key] = r_sum[key]/r_count[key]                     #计算样本均值

        return r_Q

'''
主程序
'''
if __name__ == '__main__':
    env = blackjack.BlackjackEnv()                                 #定义环境模型
    env.gamma = 1.0                                                #补充定义折扣系数
    r_Q, r_count = firstvisit_mc_actionvalue(env)                  #调用主函数
        for key, data in Q_bar.items():                            #打印结果
        print(key,": ",data)
```

运行结果如下：

```
#共 280 条数据,第 1 列为状态 - 动作对,第 2 列为该状态 - 动作对首次访问次数,第 3 列为相应的
#动作值样本均值
((19, 1, False), 0) 466.0 : -0.17381974248927037
```

```
((6, 6, False), 1) 82.0 : -0.34146341463414637
……
((5, 4, False), 1) 37.0 : -0.2972972972972973
```

代码 3-3 一共进行了 50 000 次抽样，最后产生了共 280 条动作值数据，第 1 列为状态-动作对，第 2 列为该状态-动作对首次访问次数，第 3 列为相应的动作值样本均值。其实，如果玩家手中有一个 A，则总点数根据是否将 A 看作 11 点一共有 20 种情况，如果玩家手中没有 A，则总点数为 4～21 时共有 18 种情况，庄家明牌有 A 或 2～10 共 10 种情况，所以 21 点游戏的状态空间（仅考虑了未爆煲的状态）尺度为 380；又因为玩家有叫牌和停牌两个动作，所以状态-动作对一共有 760 个，但因为在例 3-2 中有的动作状态对永远不会出现，例如((20,7,False),1)表示玩家手中总点数为 20，没有可用的 A，庄家明牌为 7 点，此时玩家继续叫牌，但根据玩家策略这是不可能的，因为总点数已经超过 18 点了，所以该状态-动作对下是不会有累积折扣奖励的，自然也就没有动作值。

【例 3-3】 用增量式每次访问蒙特卡罗策略评估完成例 3-2，代码如下：

```
##【代码 3-4】增量式每次访问蒙特卡罗策略评估算法代码

import numpy as np
import blackjack
from collections import defaultdict

'''
待评估的玩家策略:如果点数小于 18,则继续叫牌,否则停牌
'''
def simple_policy(state):
    player, dealer, ace = state
    return 0 if player >= 18 else 1                          #0:停牌;1:要牌

'''
增量式每次访问蒙特卡罗策略评估:算法 3-2 在每次访问范式下的具体实现
'''
def everyvisit_incremental_mc_actionvalue(env,num_episodes=50000):
    r_count = defaultdict(float)            #记录状态-动作对的累积折扣奖励次数
    r_Q = defaultdict(float)                                 #动作值样本均值

    #逐次采样并计算
    for i in range(num_episodes):
        #采样一条经验轨迹
        state = env.reset()                                  #环境状态初始化
        onesequence = []                                     #一条经验轨迹容器
        while True:
            action = simple_policy(state)                    #根据给定的简单策略选择动作
            next_state,reward,done,_ = env.step(action)      #交互一步
            onesequence.append((state, action, reward))      #MDP 序列
```

```
            if done:
                break
            state = next_state

        #逐个更新动作值样本均值
        for j in range(len(onesequence)):
            sa_pair = (onesequence[j][0],onesequence[j][1])
            G = sum([x[2]*np.power(env.gamma, k) for
                     k, x in enumerate(onesequence[j:])])
            r_count[sa_pair] += 1                    #记录次数
            r_Q[sa_pair] += (1.0/r_count[sa_pair])*(G-r_Q[sa_pair])

    return r_Q, r_count

'''
主程序
'''
if __name__ == '__main__':
    env = blackjack.BlackjackEnv()                   #定义环境模型
    env.gamma = 1.0                                  #补充定义折扣系数
                                                     #调用主函数
    r_Q,r_count = everyvisit_incremental_mc_actionvalue(env)
                                                     #打印结果
    for key, data in r_Q.items():
        print(key, r_count[key], ": ",data)
```

运行结果如下：

```
#共 280 条数据,第 1 列为状态-动作对,第 2 列为该状态-动作对每次访问次数,第 3 列为相应的
#动作值样本均值
((18, 3, True), 0) 75.0 : 0.1466666666666667
((11, 6, False), 1) 237.0 : 0.13924050632911392
……
((12, 4, True), 1) 19.0 : 0.2105263157894737
```

代码 3-3 和代码 3-4 的运行结果基本一致，每次访问的次数比首次访问的次数一般更多，增量式样本均值的计算方法效率更高一些。

3.2.4 蒙特卡罗和动态规划策略评估的对比

蒙特卡罗策略评估和动态规划策略评估各有优势，可适用于不同的场景，它们的优劣和区别总结如下：

（1）动态规划策略评估比蒙特卡罗策略评估更准确高效。动态规划策略评估通过求解贝尔曼方程组得到状态值或动作值，效率很高，而且理论上可以得到精确值，但蒙特卡罗策

略评估通过抽样并计算样本均值得到动作值的近似，近似程度越高，需要的样本越多，而抽样过程会导致效率降低。

(2) 蒙特卡罗策略评估比动态规划策略评估适用范围更广。动态规划策略评估是在已知状态转移概率的条件下进行的，但蒙特卡罗策略评估没有这个限制。一般来讲，能用动态规划法的问题都能适用蒙特卡罗法，反之则不然。

(3) 蒙特卡罗策略评估中动作值的计算是相互独立的，而且可以只计算一部分动作值，而动态规划策略评估中动作值之间是相互影响的，必须计算所有动作值。这是蒙特卡罗法相较于动态规划法最具特点的一个优势，当只对部分动作值感兴趣时，可以用蒙特卡罗法单独计算这些动作值，动作值之间的独立性还可以避免自举现象(Bootstrap)。这一特点在棋类对战游戏中特别重要，在对弈时，棋手一般只需考虑当前棋局应采取何种走子，对其他的大量棋局并不关心。本章第 3.5 节蒙特卡罗树搜索将对此进行延伸，但这也是蒙特卡罗法的一个天生劣势，当对所有动作值都感兴趣时，蒙特卡罗法却只能提供部分动作值，3.3 节将讨论对这一问题的解决方案。

(4) 动态规划策略评估的迭代是以时间步为单位的(Step-by-Step)，每个时间步都会更新动作值，而蒙特卡罗法的迭代是以局为单位(Episode-by-Episode)，必须完成至少一局的交互，得到一条经历完整的 MDP 序列才能更新动作值。显然，从这一点上看，动态规划更新效率更高。能否将动态规划更新效率高的优势和蒙特卡罗法免模型的优势结合呢？第 4 章将对此进行详细讨论。

3.3 蒙特卡罗强化学习

42min

蒙特卡罗策略评估还有一个问题需要解决，就是动作值的稀疏性问题。因为在抽样过程中使用的是确定性策略，所以某些状态-动作对永远不会出现在样本中，导致这些状态-动作对的动作值不存在，这又会影响到策略改进。

对这一问题有两个解决方案：一是保证每种状态-动作对都会作为初始状态-动作对出现在一些 MDP 样本序列中，这种方法称为起始探索；二是对当前策略进行一些改造，使其在产生 MDP 序列时能够保证每种状态-动作对都以一定的非零概率出现在 MDP 样本序列中，这种方法称为软策略探索。因为软策略探索涉及策略改进过程，所以将这两种策略评估的改进都放在本节来讨论。

3.3.1 蒙特卡罗策略改进

蒙特卡罗策略改进仍然使用贪婪算法，在某一状态下选择最大动作值对应的动作作为最优策略。由于状态转移概率未知，所以只能使用基于动作值函数的策略改进公式，又注意到蒙特卡罗策略评估计算的是动作值的近似，所以策略改进公式为

$$\pi'(s) \triangleq \underset{a \in \mathbf{A}}{\operatorname{argmax}} \bar{Q}_{\pi}(s,a) \tag{3-10}$$

注意到式(3-10)使用的是动作值的近似值,而非如式(2-15)的精确值,虽然理论上只要样本数趋于无穷,近似值也会收敛到精确值,但这在实际计算中当然是不现实的,那么使用近似值的策略改进方法是否依然满足策略改进定理(定理 2-1)呢? 答案是肯定的。实际上,就算是在动态规划法中,通过迭代求解贝尔曼方程组得到的解也只是动作值的近似。

3.3.2 起始探索蒙特卡罗强化学习

动作值的稀疏性是蒙特卡罗策略评估的天然缺点。稀疏性产生的原因是因为在抽样过程中使用的是确定性策略,某些状态-动作对永远不会出现在样本序列中,导致这些状态-动作对的动作值不存在。动作值的稀疏性可能会影响策略更新。

一个朴素的解决动作值稀疏性的方法是让所有状态-动作对都以一定的概率作为初始状态-动作对出现在样本序列中,这种方法叫作起始探索法(Exploring Starts)。起始探索首次访问蒙特卡罗强化学习算法的流程如下:

算法 3-3 起始探索首次访问蒙特卡罗强化学习算法

1. 输入:环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$),要生成的 MDP 序列条数 m,初始策略 π_0
2. 初始化:动作值 $\overline{Q}_\pi(s,a)=0$,随机策略 $\pi=\pi_0$
3. 过程:设置每种状态-动作对都有相同的概率出现在 MDP 序列的初始状态-动作对
4. 循环 直到前后两次策略相等
5. 随机抽样:根据当前策略 π,抽样产生 m 条经历完整的 MDP 序列
6. 策略评估:根据算法 3-1 计算 $\overline{Q}_\pi(s,a)$
7. 策略改进:根据式(3-10)进行策略改进
8. 输出:最优动作值 Q^* 和最优策略 π^*

为每种状态-动作对设置相同的初始出现概率只是起始探索法的一种实现方式,也可以使用循环所有状态-动作对,或规定状态-动作对初始出现次数等方式。

算法 3-3 使用的是策略迭代框架,也就是策略评估和策略改进交替循环。由于计算动作值样本均值之前要进行大量抽样,策略评估的效率是很低的,即使用算法 3-2 进行策略评估也不能从根本上解决这一问题。注意到 2.6 节最后对策略迭代和值迭代关系的讨论,可以将值迭代的框架引入蒙特卡罗法中。值迭代框架的关键是将策略改进过程直接嵌入策略评估的过程中,每抽样一条 MDP 序列样本都进行一次策略改进,而不是等到 m 条样本抽样完成才改进。基于值迭代框架的蒙特卡罗强化学习算法的流程如下:

算法 3-4 起始探索首次访问蒙特卡罗强化学习算法(基于值迭代框架)

1. 输入:环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$),初始贪婪策略 π_0
2. 初始化:累积回报 $G_{\text{sum}}(s,a)=0$,策略 $\pi=\pi_0$,计数器 $K(s,a)=0$
3. 过程:设置每种状态-动作对都有相同的概率出现在 MDP 序列的初始状态-动作对
4. 循环 直到策略收敛

5.　随机抽样：根据当前策略 π，抽样产生一条经历完整的 MDP 序列
6.　循环　依次遍历 MDP 序列中的所有状态-动作对(s,a)
7.　　若(s,a)首次出现在 MDP 序列中，根据式(3-5)计算累积折扣奖励 $G(s,a)$
8.　　累积回报：$G_{\text{sum}}(s,a)\leftarrow G_{\text{sum}}(s,a)+G(s,a)$
9.　　更新计数：$K(s,a)\leftarrow K(s,a)+1$
10.　　策略评估：$\overline{Q}_{\pi}(s,a)\leftarrow G_{\text{sum}}(s,a)/K(s,a)$
11.　　策略改进：根据式(3-10)进行策略改进得到新的贪婪策略
12. 输出：最优动作值 Q^{*} 和最优策略 π^{*}

细心的读者可能已经发现，在算法 3-4 中，动作值是从循环开始就累加的，不管过程中策略如何变化。这就是说，动作值的样本均值并不是当前策略下的动作值样本均值，而是历史上所有策略下的综合动作值样本均值。这是基于值迭代框架和基于策略迭代框架的蒙特卡罗强化学习法的根本区别。可以推断的是这种迭代格式不会在问题的任何非最优策略处达到(收敛)，因为如果策略稳定在某一非最优策略处，则动作值最终仍会收敛到该策略对应的动作值，也就是该策略会被准确评估。又因为策略本身并不是最优策略，所以会被准确评估得到的动作值样本均值改进，这便和循环稳定矛盾了。只有在策略和动作值都达到最优时，它们才会稳定下来，即收敛，但是算法 3-4 的严格收敛性证明仍然是强化学习领域的一个开放问题。

3.3.3　ε-贪婪策略蒙特卡罗强化学习

3.3.2 节用起始探索方法来避免样本稀疏性问题，本节介绍另一种方法：ε-贪婪策略法。

在已经介绍过的算法中，迭代过程中的策略都是确定性策略，这也正是样本稀疏性产生的根源。可以考虑用一种接近于贪婪策略的随机策略来替代贪婪策略，这样既能保证策略的贪婪性，又能保证样本的多样性，将这种策略称为软策略(Soft Policy)。

软策略是指对任意 $s\in\mathbf{S}$ 和动作 $a\in\mathbf{A}(s)$，有 $\pi(a\mid s)>0$，但渐进地收敛到贪婪策略的一种随机策略。设 π 是一个贪婪策略，基于 π 构建的一个软策略 π_{ε} 为智能体以 $1-\varepsilon$ 的概率选择当前贪婪动作，以 ε 的概率随机从所有合法动作中选择一个动作，即

$$\pi_{\varepsilon}(a\mid s)=\begin{cases}1-\varepsilon+\dfrac{\varepsilon}{|\mathbf{A}(s)|} & a\neq a^{*}\\ \dfrac{\varepsilon}{|\mathbf{A}(s)|} & a=a^{*}\end{cases}\tag{3-11}$$

这里 a^{*} 是贪婪动作，即 $a^{*}=\underset{a\in\mathbf{A}(s)}{\operatorname{argmax}}\pi(a\mid s)$，称策略 π_{ε} 为策略 π 的 ε-贪婪策略。显然，ε-贪婪策略是一个软策略，而且是对应于贪婪策略 π 的所有软策略中最容易操作的一个。以下考虑用 ε-贪婪策略来设计蒙特卡罗强化学习算法。

首先要解决的问题是，如果策略改进时使用 ε-贪婪策略，则迭代过程是否能满足策略改进

定理(定理 2-1)呢？答案是肯定的。设原策略是 π，改进后的 ε-贪婪策略是 π'，则对 $\forall s\in\mathbf{S}$

$$\begin{aligned}
Q_\pi(s,\pi'(s)) &= E_{\mathbf{A}\sim\pi'(\cdot|s)}[Q_\pi(s,A)] \\
&= \sum_{a\in\mathbf{A}}\pi'(a\mid s)Q_\pi(s,a) \\
&= \frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}Q_\pi(s,a)+(1-\varepsilon)\max_{a\in\mathbf{A}}Q_\pi(s,a) \\
&\geqslant \frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}Q_\pi(s,a)+(1-\varepsilon)\sum_{a\in\mathbf{A}}\frac{\pi(a\mid s)-\frac{\varepsilon}{|\mathbf{A}|}}{1-\varepsilon}Q_\pi(s,a) \\
&= \frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}Q_\pi(s,a)-\frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}Q_\pi(s,a)+\sum_{a\in\mathbf{A}}\pi(a\mid s)Q_\pi(s,a) \\
&= V_\pi(s)
\end{aligned} \tag{3-12}$$

这里$\geqslant$是因为

$$\sum_{a\in\mathbf{A}}\frac{\pi(a\mid s)-\frac{\varepsilon}{|\mathbf{A}|}}{1-\varepsilon}=\frac{1}{1-\varepsilon}\left[\sum_{a\in\mathbf{A}}\pi(a\mid s)-\sum_{a\in\mathbf{A}}\frac{\varepsilon}{|\mathbf{A}|}\right]=\frac{1}{1-\varepsilon}(1-\varepsilon)=1$$

也就是说，

$$\sum_{a\in\mathbf{A}}\frac{\pi(a\mid s)-\frac{\varepsilon}{|\mathbf{A}|}}{1-\varepsilon}Q_\pi(s,a)$$

是所有 $Q_\pi(s,a)$ 在分布律 $\mathbf{A}\sim(\pi(\cdot\mid s)-\varepsilon/|\mathbf{A}|)/(1-\varepsilon)$ 下的期望，它自然小于最大的 $Q_\pi(s,a)$，即 $\max\limits_{a\in\mathbf{A}}Q_\pi(s,a)$。

式(3-12)说明用 ε-贪婪策略来改进策略是可以满足策略改进条件(2-14)的。

接下来要解决的是收敛性问题。假设 V^* 和 Q^* 分别是 ε-贪婪策略下的最优状态和动作值，根据定义，V^* 应该满足方程

$$\begin{aligned}
V^*(s) &= (1-\varepsilon)\max_{a\in\mathbf{A}}Q^*(s,a)+\frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}Q^*(s,a) \\
&= (1-\varepsilon)\max_{a\in\mathbf{A}}\sum_{s'\in\mathbf{S}}p(s,a,s')[r+\gamma V^*(s')]+ \\
&\quad \frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}\sum_{s'\in\mathbf{S}}p(s,a,s')[r+\gamma V^*(s')]
\end{aligned} \tag{3-13}$$

也就是说，式(3-13)是最优解应该满足的条件。

在式(3-12)中，若 π 和 π' 相同，则

$$\begin{aligned}
Q_\pi(s,\pi'(s)) &= \sum_{a\in\mathbf{A}}\pi'(a\mid s)Q_\pi(s,a) \\
&= \sum_{a\in\mathbf{A}}\pi(a\mid s)Q_\pi(s,a) \\
&= V_\pi(s)
\end{aligned} \tag{3-14}$$

说明式(3-12)中≥中的＝成立，这时显然有

$$\begin{aligned} V_\pi(s) &= (1-\varepsilon)\max_{a\in\mathbf{A}} Q_\pi(s,a) + \frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}} Q_\pi(s,a) \\ &= (1-\varepsilon)\max_{a\in\mathbf{A}} \sum_{s'\in\mathbf{S}} p(s,a,s')[r+\gamma V_\pi(s')] + \\ &\quad \frac{\varepsilon}{|\mathbf{A}|}\sum_{a\in\mathbf{A}}\sum_{s'\in\mathbf{S}} p(s,a,s')[r+\gamma V_\pi(s')] \end{aligned} \tag{3-15}$$

式(3-15)和式(3-13)除状态、动作值符号不同外完全一样，说明 π 是满足最优性条件的。也就是说，在使用 ε-贪婪策略进行动作选择的设定下，只要前后两次迭代得到的贪婪策略是相同的，就可以保证策略已经达到最优。

在策略已经达到最优($\pi=\pi'=\pi^*$)时，式(3-15)和式(3-13)其实是一样的，而且如果将式(3-15)中的 π 换成 π'，得到的方程也和式(3-13)一样，所以

$$V_\pi(s) = V_{\pi'}(s) = V^*(s) \tag{3-16}$$

综上所述，当前后两次策略或状态值相同时，策略和状态值都达到最优，这就是迭代过程的终止条件。

基于上述分析，可以总结出基于值迭代框架的 ε-贪婪策略首次访问蒙特卡罗强化学习算法的流程如下：

算法 3-5　ε-贪婪策略首次访问蒙特卡罗强化学习算法(基于值迭代框架)

1. 输入：环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$)，初始贪婪策略 π_0
2. 初始化：累积回报 $G_{\text{sum}}(s,a)=0$，策略 $\pi=\pi_0$，计数器 $K(s,a)=0$
3. 过程：(设置每种状态-动作对都有相同的概率出现在 MDP 序列的初始状态-动作对)
4. 　　循环　直到两次策略相同
5. 　　贪婪策略：根据式(3-11)改造策略 π，构造 ε-贪婪策略 π_ε
6. 　　随机抽样：根据策略 π_ε，抽样产生一条经历完整的 MDP 序列
7. 　　循环　依次遍历 MDP 序列中的所有状态-动作对(s,a)
8. 　　若(s,a)首次出现在 MDP 序列中，则根据式(3-5)计算累积折扣奖励 $G(s,a)$
9. 　　累积回报：$G_{\text{sum}}(s,a)\leftarrow G_{\text{sum}}(s,a)+G(s,a)$
10. 　　更新计数：$K(s,a)\leftarrow K(s,a)+1$
11. 　　策略评估：$\overline{Q}(s,a)\leftarrow G_{\text{sum}}(s,a)/K(s,a)$
12. 　　策略改进：根据式(3-10)进行策略改进得到新的贪婪策略 π
13. 输出：最优动作值 Q^* 和最优策略 π^*

为了保证每种状态-动作对都能被访问，可以将起始探索和软策略探索结合起来使用。

3.3.4　蒙特卡罗强化学习案例

【例 3-4】 用基于值迭代的起始探索每次访问蒙特卡罗强化学习算法，寻找 21 点问题的最优策略，算法的代码如下：

```
##【代码 3-5】基于值迭代的起始探索每次访问蒙特卡罗强化学习算法

import numpy as np
import blackjack
from collections import defaultdict
import matplotlib.pyplot as plt

'''
基于值迭代的起始探索每次访问蒙特卡罗强化学习算法类
'''
class StartExplore_EveryVisit_ValueIter_MCRL():
    ##类初始化
    def __init__(self,env,num_episodes=10000):
        self.env = env
        self.nA = env.action_space.n                          #动作空间尺度
                                                              #动作值函数
        self.r_Q = defaultdict(lambda: np.zeros(self.nA))
                                                              #累积折扣奖励之和
        self.r_sum = defaultdict(lambda: np.zeros(self.nA))
                                                              #累积折扣奖励次数
        self.r_cou = defaultdict(lambda: np.zeros(self.nA))
        self.policy = defaultdict(int)                        #各状态下的策略
        self.num_episodes = num_episodes                      #最大抽样回合数

    ##策略初始化及改进函数,如果初始化为点数小于 18,则继续叫牌,否则停牌
    def update_policy(self,state):
        if state not in self.policy.keys():
            player, dealer, ace = state
            action = 0 if player >= 18 else 1                 #0:停牌;1:要牌
        else:
                                                              #最优动作值对应的动作
            action = np.argmax(self.r_Q[state])

        self.policy[state] = action

    ##蒙特卡罗抽样产生一条经历完整的 MDP 序列
    def mc_sample(self):
        onesequence = []                                      #经验轨迹容器

        #基于贪婪策略产生一条轨迹
        state = self.env.reset()                              #起始探索产生初始状态
        while True:
            self.update_policy(state)                         #策略改进
            action = self.policy[state]                       #根据策略选择动作
            next_state,reward,done,_ = env.step(action)       #交互一步
            onesequence.append((state,action,reward))         #经验轨迹
```

```
            state = next_state
            if done:                                    #游戏是否结束
                break

        return onesequence

    ##蒙特卡罗每次访问策略评估一条序列
    def everyvisit_valueiter_mc(self,onesequence):
        #访问经验轨迹中的每种状态-动作对
        for k, data_k in enumerate(onesequence):
            state = data_k[0]                           #状态
            action = data_k[1]                          #动作
            #计算累积折扣奖励
            G = sum([x[2]*np.power(env.gamma,i) for i, x
                        in enumerate(onesequence[k:])])
            self.r_sum[state][action] += G              #累积折扣奖励之和
            self.r_cou[state][action] += 1.0            #累积折扣奖励次数
            self.r_Q[state][action] = self.r_sum[
                    state][action]/self.r_cou[state][action]

    ##蒙特卡罗强化学习
    def mcrl(self):
        for i in range(self.num_episodes):
            #起始探索抽样一条 MDP 序列
            onesequence = self.mc_sample()
            #值迭代过程,结合了策略评估和策略改进
            self.everyvisit_valueiter_mc(onesequence)

        opt_policy = self.policy                        #最优策略
        opt_Q = self.r_Q                                #最优动作值

        return opt_policy, opt_Q

    ##绘制最优策略图像
    def draw(self,policy):
        true_hit = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == True and policy[x] == 1]
        true_stick = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == True and policy[x] == 0]
        false_hit = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == False and policy[x] == 1]
        false_stick = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == False and policy[x] == 0]

        plt.figure(1)
        plt.plot([x[0] for x in true_hit],
```

```
                [x[1] for x in true_hit],'bo',label = 'HIT')
        plt.plot([x[0] for x in true_stick],
                [x[1] for x in true_stick],'rx',label = 'STICK')
        plt.xlabel('dealer'), plt.ylabel('player')
        plt.legend(loc = 'upper right')
        plt.title('Usable Ace')
        filepath = 'code3 - 5 UsabelAce.png'
        plt.savefig(filepath, dpi = 300)

        plt.figure(2)
        plt.plot([x[0] for x in false_hit],
                [x[1] for x in false_hit],'bo',label = 'HIT')
        plt.plot([x[0] for x in false_stick],
                [x[1] for x in false_stick],'rx',label = 'STICK')
        plt.xlabel('dealer'), plt.ylabel('player')
        plt.legend(loc = 'upper right')
        plt.title('No Usable Ace')
        filepath = 'code3 - 5 NoUsabelAce.png'
        plt.savefig(filepath, dpi = 300)

'''
主程序
'''
if __name__ == '__main__':
    env = blackjack.BlackjackEnv()                    #导入环境模型
    env.gamma = 1                                     #补充定义折扣系数

    agent = StartExplore_EveryVisit_ValueIter_MCRL(
            env,num_episodes = 1000000)
    opt_policy,opt_Q = agent.mcrl()
    for key in opt_policy.keys():
        print(key,": ",opt_policy[key],opt_Q[key])
    agent.draw(opt_policy)
```

运行结果如下：

```
(5, 4, False) : 1 [ - 1.          - 0.16497175]
(4, 2, False) : 0 [ - 0.36091954   - 1.       ]
…
(12, 8, True) : 0 [ - 0.53456221   - 1.       ]
```

运行结果中第 1 列为状态，第 2 列为相应状态下的最优动作，第 3 列为相应状态下的所有动作值的均值，按动作编号从小到大排列。所有运行结果生成的策略如图 3-2 所示。

【例 3-5】 用基于值迭代的 ε-贪婪策略每次访问蒙特卡罗强化学习算法，寻找 21 点问题的最优策略，算法代码如下：

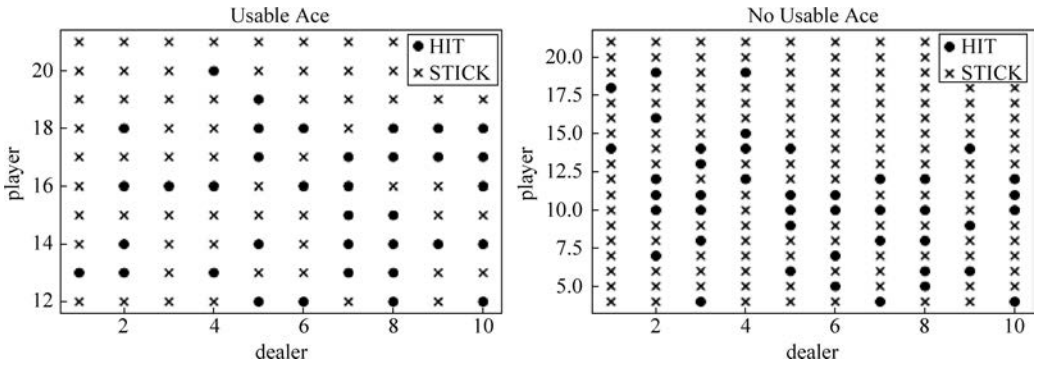


图 3-2　基于值迭代的起始探索每次访问蒙特卡罗强化学习求解 21 点问题策略示意图

```
##【代码 3-6】基于值迭代的 ε-贪婪策略每次访问蒙特卡罗强化学习算法代码

import numpy as np
import blackjack
from collections import defaultdict
import matplotlib.pyplot as plt

'''
基于值迭代的 epsilon-贪婪策略每次访问蒙特卡罗强化学习算法类
'''
class SoftExplore_EveryVisit_ValueIter_MCRL():
    ##类初始化
    def __init__(self,env,num_episodes=10000,epsilon=0.1):
        self.env = env
        self.nA = env.action_space.n                          #动作空间维度
        self.Q_bar = defaultdict(lambda: np.zeros(self.nA))   #动作值函数
        self.G_sum = defaultdict(lambda: np.zeros(self.nA))   #累积折扣奖励之和
        self.G_cou = defaultdict(lambda: np.zeros(self.nA))   #累积折扣奖励次数
        self.g_policy = defaultdict(int)                      #贪婪策略
                                                              #epsilon 策略
        self.eg_policy = defaultdict(lambda: np.zeros(self.nA))
        self.num_episodes = num_episodes                      #最大抽样回合数
        self.epsilon = epsilon

    ##策略初始化及改进函数,如果初始化为点数小于 18,则继续叫牌,否则停牌
    def update_policy(self,state):
        if state not in self.g_policy.keys():
            player, dealer, ace = state
            action = 0 if player >= 18 else 1                 #0:停牌;1:要牌
        else:
            action = np.argmax(self.Q_bar[state])             #最优动作值对应的动作

        #贪婪策略
```

```
        self.g_policy[state] = action
        #对应的 epsilon-贪婪策略
        self.eg_policy[state] = np.ones(self.nA) * self.epsilon/self.nA
        self.eg_policy[state][action] += 1 - self.epsilon

        return self.g_policy[state], self.eg_policy[state]

    ##蒙特卡罗抽样产生一条经历完整的 MDP 序列
    def mc_sample(self):
        onesequence = []                                    #经验轨迹容器

        #基于 epsilon-贪婪策略产生一条轨迹
        state = self.env.reset()                            #初始状态
        while True:
            _,action_prob = self.update_policy(state)
            action = np.random.choice(np.arange(len(action_prob)),
                                      p = action_prob)
            next_state,reward,done,info = env.step(action)  #交互一步
            onesequence.append((state,action,reward,info))  #经验轨迹
            state = next_state
            if done:                                        #游戏是否结束
                break

        return onesequence

    ##蒙特卡罗每次访问策略评估一条序列
    def everyvisit_valueiter_mc(self,onesequence):
        #访问经验轨迹中的每种状态-动作对
        for k, data_k in enumerate(onesequence):
            state = data_k[0]
            action = data_k[1]
            #计算累积折扣奖励
            G = sum([x[2] * np.power(env.gamma,i) for i, x
                     in enumerate(onesequence[k:])])
            self.G_sum[state][action] += G                  #累积折扣奖励之和
            self.G_cou[state][action] += 1.0                #累积折扣奖励次数
            self.Q_bar[state][action] = self.G_sum[
                      state][action]/self.G_cou[state][action]

    ##蒙特卡罗强化学习
    def mcrl(self):
        for i in range(self.num_episodes):
            #起始探索抽样一条 MDP 序列
            onesequence = self.mc_sample()

            #值迭代过程,结合了策略评估和策略改进
```

```
            self.everyvisit_valueiter_mc(onesequence)

        opt_policy = self.g_policy                          #最优策略
        opt_Q = self.Q_bar                                  #最优动作值

        return opt_policy, opt_Q

    ##绘制最优策略图像
    def draw(self,policy):
        true_hit = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == True and policy[x] == 1]
        true_stick = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == True and policy[x] == 0]
        false_hit = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == False and policy[x] == 1]
        false_stick = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == False and policy[x] == 0]

        plt.figure(1)
        plt.plot([x[0] for x in true_hit],
                 [x[1] for x in true_hit],'bo',label = 'HIT')
        plt.plot([x[0] for x in true_stick],
                 [x[1] for x in true_stick],'rx',label = 'STICK')
        plt.xlabel('dealer'), plt.ylabel('player')
        plt.legend(loc = 'upper right')
        plt.title('Usable Ace')
        filepath = 'code3-6 UsabelAce.png'
        plt.savefig(filepath, dpi = 300)

        plt.figure(2)
        plt.plot([x[0] for x in false_hit],
                 [x[1] for x in false_hit],'bo',label = 'HIT')
        plt.plot([x[0] for x in false_stick],
                 [x[1] for x in false_stick],'rx',label = 'STICK')
        plt.xlabel('dealer'), plt.ylabel('player')
        plt.legend(loc = 'upper right')
        plt.title('No Usable Ace')
        filepath = 'code3-6 NoUsabelAce.png'
        plt.savefig(filepath, dpi = 300)

'''
主程序
'''
if __name__ == '__main__':
    env = blackjack.BlackjackEnv()                          #导入环境模型
    env.gamma = 1                                           #补充定义折扣系数
```

```
    agent = SoftExplore_EveryVisit_ValueIter_MCRL(
            env,num_episodes = 1000000,epsilon = 0.1)
    opt_policy,opt_Q = agent.mcrl()
    for key in opt_policy.keys():
        print(key,": ",opt_policy[key],opt_Q[key])
    agent.draw(opt_policy)
```

运行结果如下：

```
(13, 1, True) : [0. 1.]  [-0.85714286  -0.35135135]
(4, 7, False) : [1. 0.]  [-0.35135135  -1. ]
……
(12, 1, True) : [1. 0.]  [-0.68  -1. ]
```

由结果生成的策略如图 3-3 所示。

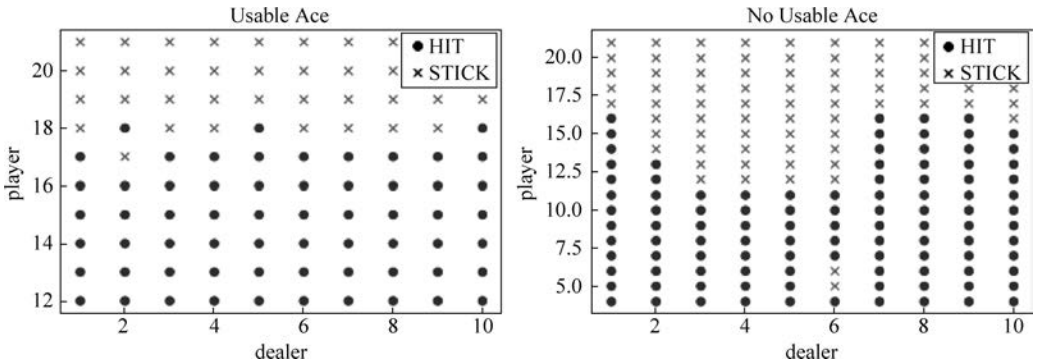


图 3-3　基于值迭代的软策略探索每次访问蒙特卡罗强化学习求解 21 点问题策略示意图

比较图 3-2 和图 3-3 可以看出，基于 ε-贪婪策略的软策略探索比起始探索取得了更加稳定的最终策略。以上两例都是基于值迭代框架的，读者可以练习编写基于策略迭代框架的对应代码。

3.4　异策略蒙特卡罗强化学习

所有的强化学习算法都面临着一个困境：强化学习的目标是寻找最优策略，但学习过程需要按照非最优策略来产生足够多的具备探索性能的经验数据以尽量覆盖对策略空间的搜索。我们将待学习的策略称为目标策略(Target Policy)，将用于产生经验数据的策略称为行为策略(Behavior Policy)。

如果目标策略和行为策略相同，则称为同策略(On-policy)强化学习。在同策略强化学习中，上述困境显得尤为明显，因为随着策略逐渐收敛，按照策略产生的经验数据就会逐渐

变得单一，不具有探索性能，这时策略就可能停留在一个次最优状态，无法改进。一个直截了当的解决方案就是让目标策略和行为策略不同，目标策略只负责学习最优策略，而行为策略只负责产生经验数据，这种方法称为异策略(Off-policy)强化学习，但异策略强化学习又会产生新的问题，因为策略评估的对象是目标策略，但用于评估的经验数据却是按行为策略产生的，这显然不行。解决这一问题的方案是概率中经常使用的重要性采样。

本节首先介绍重要性采样原理，然后介绍异策略蒙特卡罗策略评估，最后介绍异策略蒙特卡罗强化学习。

3.4.1　重要性采样

本节用蒙特卡罗法近似计算定积分的例子来介绍重要性采样的必要性和原理。我们知道，用蒙特卡罗法计算区间$[a,b]$上$f(x)$的定积分的步骤如下。

步骤1：在区间$[a,b]$上按均匀分布采样$n+1$个点：$\{x_0,x_1,x_2,\cdots,x_n\}$，其中$x_0=a$，$x_n=b$。

步骤2：计算样本点处的函数值：$\{f(x_1),f(x_2),\cdots,f(x_n)\}$。

步骤3：定积分的估计式为

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i) \tag{3-17}$$

这里$(b-a)/n$相当于第i个长方形的宽，而$f(x_i)$相当于第i个长方形的高，如图3-4所示。

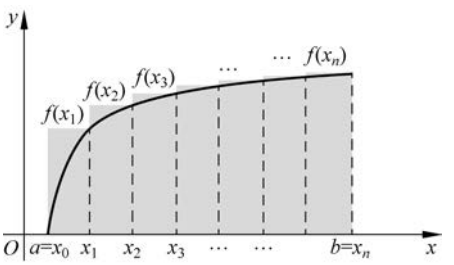


图3-4　蒙特卡罗法估计定积分示意图

上述估计方法使用均匀分布进行采样，估计精度会随着采样点数的增加而越发准确，但如何在采样数一定的情况下让估计尽可能准确呢？这就需要对采样的概率进行干预，如图3-5(a)所示，A部分函数值变化较快，B部分较慢，显然应该让A部分采样更加密集，B部分采样更加稀疏，于是可以按图3-5(b)所示的概率密度函数$p(x)$进行采样，但这样就不能再用式(3-17)来估计定积分了，因为采样是不均匀的，小矩形的宽不等长，所以要对其进行加权，这个权重就是重要性权重。

其实，函数$f(x)$在区间$[a,b]$上的积分可以看作$f(x)$在某一分布下的期望，即

$$\int_a^b f(x)\mathrm{d}x \triangleq E_{X\sim\pi(\cdot)}[f(x)] \tag{3-18}$$

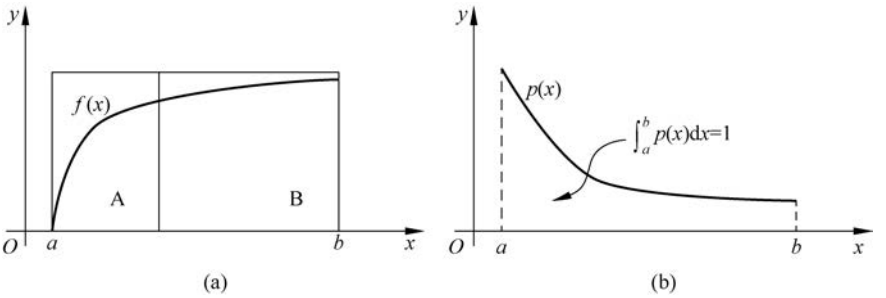


图 3-5　重要性采样估计定积分示意图

只是分布 π 是不知道的，自然不能用基于 π 的抽样来估计定积分，但

$$\begin{aligned}\int_a^b f(x)\mathrm{d}x &\triangleq E_{X\sim\pi(\cdot)}[f(x)]\\ &=\int_a^b \pi(x)f(x)\mathrm{d}x\\ &=\int_a^b p(x)\frac{\pi(x)}{p(x)}f(x)\mathrm{d}x\\ &=E_{X\sim p(\cdot)}\left[\frac{\pi(x)}{p(x)}f(x)\right]\\ &\approx\frac{b-a}{n}\sum_{i=1}^{n}\frac{\pi(x_i)}{p(x_i)}f(x_i)\end{aligned}\tag{3-19}$$

式(3-19)说明函数 $f(x)$ 在区间 $[a,b]$ 上的定积分也可以看作函数 $(\pi(x)/p(x))f(x)$ 在分布 $p(x)$ 下的期望，而 $\pi(x)/p(x)$ 就是重要性权重；最后一个约等式是其估计值，这一估计和式(3-17)不同的是其样本是在分布 $p(x)$ 下抽样得到的，而不是按均匀分布抽样的。

当然，式(3-19)仍不能计算，因为 $\pi(x)$ 是未知的，若 $\pi(x)$ 已知，但并不希望按 $\pi(x)$ 来抽样估计定积分，而是希望按另一个分布 $p(x)$ 来抽样估计时，式(3-19)就可以使用了。本章的异策略强化学习就是这样使用的。

3.4.2　异策略蒙特卡罗策略评估

异策略蒙特卡罗策略评估的基本思想是使用按行为策略采样产生的经验轨迹来评估目标策略，这里行为策略和目标策略是不同的。使用重要性采样进行策略评估，首先讨论重要性权重。

设学习过程从状态 s_t 出发，按照任意策略 π 选择动作，产生一条经验轨迹

$$s_t,a_t,R_{t+1},s_{t+1},a_{t+1},R_{t+2},\cdots,s_{T-1},a_{T-1},R_T,s_T$$

其中，s_T 是终止状态，则产生该条经验轨迹的概率为

$$\begin{aligned}&\Pr(s_t,a_t,\cdots,s_{T-1},a_{T-1},s_T)\\ &=\pi(a_t\mid s_t)p(s_{t+1}\mid s_t,a_t)\pi(a_{t+1}\mid s_{t+1})\cdots\pi(a_{T-1}\mid s_{T-1})p(s_T\mid s_{T-1},a_{T-1})\end{aligned}$$

$$=\prod_{k=t}^{T-1}\pi(a_k \mid s_k)p(s_{k+1} \mid s_k,a_k) \tag{3-20}$$

根据重要性采样原理，若按照行为策略 b 进行采样产生一条经验轨迹，则该样本相较于目标策略 π 的重要性权重为

$$\rho=\frac{\prod_{k=t}^{T-1}\pi(a_k \mid s_k)p(s_{k+1} \mid s_k,a_k)}{\prod_{k=t}^{T-1}b(a_k \mid s_k)p(s_{k+1} \mid s_k,a_k)}=\prod_{k=t}^{T-1}\frac{\pi(a_k \mid s_k)}{b(a_k \mid s_k)} \tag{3-21}$$

可以看出，虽然样本产生的概率依赖于策略和状态转移概率，但重要性权重却只依赖于策略。

有了重要性权重，就可以用按行为策略 b 产生的经验数据来评估目标策略 π 了。过程和 3.2.1 节首次访问蒙特卡罗策略评估的过程一样，唯一不同的是经验数据是按行为策略 b 而非目标策略 π 产生的，故式(3-7)应改写为

$$\overline{Q}_\pi(s,a)=\frac{\sum_{i=1}^{m}\rho^{<i>}(s,a)G^{<i>}(s,a)}{\sum_{i=1}^{m}I^{<i>}(s,a)} \tag{3-22}$$

其中

$$\rho^{<i>}(s,a)=\prod_{k=t}^{T_i-1}\left(\frac{\pi(a_k^{<i>} \mid s_k^{<i>})}{b(a_k^{<i>} \mid s_k^{<i>})} \mid s_t^{<i>}=s,a_t^{<i>}=a\right) \tag{3-23}$$

为第 i 条经验轨迹首次访问状态-动作对(s,a)的重要性权重，$I^{<i>}(s,a)$为第 i 条经验轨迹首次访问状态-动作对(s,a)的指示函数。

式(3-22)用算术平均来估计动作值，称为一般重要性采样估计(Ordinary Importance Sampling Estimation)。也可以用加权平均来估计，即

$$\overline{Q}_\pi(s,a)=\frac{\sum_{i=1}^{m}\rho^{<i>}(s,a)G^{<i>}(s,a)}{\sum_{i=1}^{m}\rho^{<i>}(s,a)} \tag{3-24}$$

称为加权重要性采样估计(Weighted Importance Sampling Estimation)。

一般重要性采样估计和加权重要性采样估计是两种典型的重要性采样估计方法，可以通过估计的偏置和方差来评估它们的优劣。一般重要性采样估计是无偏的，而加权重要性采样估计是有偏的，但其偏置可以收敛到 0。一般重要性采样估计的方差是无界的，因为重要性权重是无界的，但加权重要性采样估计的方差可以收敛到 0，即使是在重要性权重无界的情况下，所以在实际计算中，加权重要性采样估计更受青睐。

而实际上，由于重要性采样比率涉及所有状态的转移概率，因此有很高的方差，从这一点来讲，蒙特卡罗算法不太适合于处理异策略问题。异策略蒙特卡罗强化学习只有理论研究价值，实际应用效果并不明显，难以获得最优动作值函数。

值得注意的是，以上估计公式和分析都是建立在首次访问基础上的。对于每次访问模式，一般和加权重要性采样估计都是有偏的，但加权重要性采样估计的偏置仍可以收敛到0。因为每次访问模式比首次访问模式更容易编程实现，所以在实际计算中，一般使用基于每次访问的加权重要性采样估计。

异策略每次访问加权重要性采样蒙特卡罗策略评估算法的流程如下：

算法 3-6　异策略每次访问加权重要性采样蒙特卡罗策略评估算法

1. 输入：环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$)，目标策略 π，行为策略 b，MDP 序列数 m
2. 初始化：累积回报 $G_{\text{sum}}(s,a)=0$，累积权重 $\rho_{\text{sum}}(s,a)=0$
3. 过程：
4. 　　随机抽样：根据行为策略 b，抽样产生 m 条经历完整的 MDP 序列
5. 　　循环：$i=1\sim m$
6. 　　循环：依次遍历第 i 条 MDP 序列中的所有状态-动作对(s,a)
7. 　　(s,a)每次出现在 MDP 序列中时，根据式(3-5)计算累积折扣奖励
8. 　　根据式(3-23)计算重要性权重
9. 　　累积回报：$G_{\text{sum}}(s,a)\leftarrow G_{\text{sum}}(s,a)+\rho^{<i>}(s,a)G^{<i>}(s,a)$
10. 　　累积权重：$\rho_{\text{sum}}(s,a)\leftarrow\rho_{\text{sum}}(s,a)+\rho^{<i>}(s,a)$
11. 　　策略评估：根据式(3-24)评估目标策略
12. 输出：样本均值 $\overline{Q}_{\pi}$ 作为动作值的近似

3.4.3　增量式异策略蒙特卡罗策略评估

可以将 3.2.2 节的增量式样本均值计算范式推广到异策略蒙特卡罗策略评估。对于一般重要性采样估计式(3-22)的推广是显然的，和式(3-9)完全一致。以下讨论对加权重要性采样估计式(3-24)的推广。

由式(3-24)有

$$\overline{Q}_{\pi}^{<k>}(s,a)=\frac{\sum_{i=1}^{k}\rho^{<i>}(s,a)G^{<i>}(s,a)}{\sum_{i=1}^{k}\rho^{<i>}(s,a)}$$

$$=\frac{\sum_{i=1}^{k-1}\rho^{<i>}(s,a)G^{<i>}(s,a)+\rho^{<k>}(s,a)G^{<k>}(s,a)}{\sum_{i=1}^{k}\rho^{<i>}(s,a)}$$

$$=\frac{\overline{Q}_{\pi}^{<k-1>}(s,a)\sum_{i=1}^{k-1}\rho^{<i>}(s,a)+\rho^{<k>}(s,a)G^{<k>}(s,a)}{\sum_{i=1}^{k}\rho^{<i>}(s,a)}$$

$$\begin{aligned}
&\bar{Q}_{\pi}^{<k-1>}(s,a)\sum_{i=1}^{k}\rho^{<i>}(s,a)+\rho^{<k>}(s,a)G^{<k>}(s,a)-\rho^{<k>}(s,a)\bar{Q}_{\pi}^{<k-1>}(s,a)\\
=&\frac{}{\sum_{i=1}^{k}\rho^{<i>}(s,a)}\\
=&\bar{Q}_{\pi}^{<k-1>}(s,a)+\frac{\rho^{<k>}(s,a)}{\sum_{i=1}^{k-1}\rho^{<i>}(s,a)+\rho^{<k>}(s,a)}(G^{<k>}(s,a)-\bar{Q}_{\pi}^{<k-1>}(s,a))
\end{aligned}\tag{3-25}$$

式(3-25)的结果可简写成

$$\begin{aligned}
C^{<k>}&=C^{<k-1>}+\rho^{<k>}\\
\bar{Q}^{<k>}&=\bar{Q}^{<k-1>}+\frac{\rho^{<k>}}{C^{<k>}}(G^{<k>}-\bar{Q}^{<k-1>})
\end{aligned}\tag{3-26}$$

式(3-26)就是加权重要性采样估计的增量形式。

算法 3-6 首先生成所有的经验样本，再进行评估，使用增量形式可以每生成一条样本就进行一次评估，直到终止条件满足为止。这种方式是后文基于值迭代框架的异策略蒙特卡罗强化学习算法的基础。基于增量式样本均值计算范式的异策略每次访问加权重要性采样蒙特卡罗策略评估算法的流程如下：

算法 3-7　增量式异策略每次访问加权重要性采样蒙特卡罗策略评估算法

1. 输入：环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$)，目标策略 π，行为策略 b，最大迭代局数 m
2. 初始化：动作值 $\bar{Q}_{\pi}(s,a)=0$，回报 $G(s,a)=0$，累积重要性权重 $\rho_{\text{sum}}(s,a)=0$，重要性权重 $\rho(s,a)=0$
3. 过程：
4. 　　循环：episode$=1\sim m$
5. 　　随机抽样：根据行为策略 b，抽样产生一条经历完整的 MDP 序列
6. 　　循环：依次遍历 MDP 序列中的所有状态-动作对(s,a)
7. 　　根据式(3-5)计算累积折扣奖励 $G(s,a)$
8. 　　根据式(3-23)计算重要性权重 $\rho(s,a)$
9. 　　累积权重：$\rho_{\text{sum}}(s,a)\leftarrow\rho_{\text{sum}}(s,a)+\rho(s,a)$
10. 　　策略评估：根据式(3-26)计算 $\bar{Q}_{\pi}(s,a)$
11. 输出：动作值估计 $\bar{Q}_{\pi}(s,a)$

3.4.4　异策略蒙特卡罗强化学习

用贪婪策略作为目标策略，用任意随机策略作为行为策略，可以得到用于估算最优贪婪策略的异策略蒙特卡罗强化学习算法，其算法流程如下：

算法 3-8　异策略蒙特卡罗强化学习算法

1. 输入：环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$)，最大迭代局数 m

2. 初始化：动作值 $\bar{Q}_{\pi}(s,a)=0$，回报 $G(s,a)=0$，累积重要性权重 $\rho_{\text{sum}}(s,a)=0$，初始化目标策略 $\pi(s)\leftarrow\arg\max_{a}Q(s,a)$

3. 过程：

4. 　　循环 episode=1～m

5. 　　生成随机行为策略 b

6. 　　随机抽样：根据行为策略 b，抽样产生一条经历完整的 MDP 序列

7. 　　$G\leftarrow 0$

8. 　　$\rho\leftarrow 1$

9. 　　循环　自后向前遍历 MDP 序列：$t=T-1,T-2,\cdots,0$

10. 　　累积折扣奖励：$G\leftarrow\gamma G+R_{t+1}$

11. 　　累积权重：$\rho_{\text{sum}}(s_t,a_t)\leftarrow\rho_{\text{sum}}(s_t,a_t)+\rho$

12. 　　策略评估：$\bar{Q}(s_t,a_t)\leftarrow\bar{Q}(s_t,a_t)+(\rho/\rho_{\text{sum}}(s_t,a_t))(G-\bar{Q}(s_t,a_t))$

13. 　　策略改进：$\pi(s_t)\leftarrow\arg\max_{a}Q(s_t,a)$

14. 　　如果 $a_t\neq\pi(s_t)$，则结束循环，进入下一序列

15. 　　权重更新：$\rho\leftarrow\rho\cdot(\pi(s_t|a_t)/b(s_t|a_t))$

16. 输出：最优动作值 $\bar{Q}^{*}(s,a)$，最优策略 $\pi^{*}(s)$

关于算法 3-8 的几点说明如下：

(1) 在内层循环中，遍历一条 MDP 序列的方向是自后向前而非自前向后的，这主要出于提高计算效率减少重复计算的考虑。因为自前向后遍历计算累积折扣奖励和重要性权重时，会存在重复计算，而自后向前遍历的循环可以避免这一点。

(2) 在算法 3-8 中，极有可能出现的一种情况是 $a_t\neq\pi(s_t)$，内层循环被迫提前终止，进入下一条 MDP 序列。目标策略是贪婪策略，而行为策略是软策略，所以按行为策略产生的 MDP 序列极有可能和目标策略期望产生的 MDP 序列不一致，这时重要性权重等于 0，继续循环就失去了意义，所以必须废弃剩下的状态-动作对，而直接启用一条新的 MDP 序列。这就意味着，大部分的策略评估只发生在 MDP 序列靠尾部的状态-动作对中，这种现象称为尾部学习效应(Learning on the Tail)。尾部学习效应会大大降低异策略强化学习的效率，是异策略强化学习算法最大的一个困难，也是异策略强化学习虽然理论漂亮，但实用性欠佳的根源。目前没有比较好的办法来解决尾部学习效应问题，一个基本可行的解决方案是将时序差分引入蒙特卡罗强化学习，这将在第 4 章详细介绍。

(3) 由于贪婪策略是确定性策略，即当 $a_t=\pi(s_t)$ 时，$\pi(s_t|a_t)=1$，所以权重更新公式也可以写作

$$\rho\leftarrow\rho\cdot\frac{1}{b(s_t\mid a_t)}$$

(4) 可以使用贪婪策略 π 对应的 ε-贪婪策略作为行为策略，即 $b=\pi_{\varepsilon}$。通过控制 ε 的大

小可以控制目标策略和行为策略的异化程度，ε 越大($0 \leqslant \varepsilon \leqslant 1$)，行为策略的随机性越强，算法的探索能力越强，但尾部学习效应也越明显。特别地，当 $\varepsilon = 0$ 时，目标策略和行为策略相同，算法 3-8 退化为同策略算法；当 $\varepsilon = 1$ 时，行为策略完全随机，此时探索算法的探索能力最强，但尾部学习效应非常明显。

3.4.5　异策略蒙特卡罗强化学习案例

【例 3-6】 以 21 点游戏(例 3-2)为例来讨论一般重要性采样估计和加权重要性采样估计的区别。

由前文可知，蒙特卡罗策略评估可以单独评估某种状态在任意策略下的动作值。假设待评估的目标策略为玩家总点数大于或等于 18 则停牌，否则继续叫牌，而待评估的状态-动作对为((13,2,True),1)，即玩家总点数为 13，有一张可用的 A，庄家明牌为 2，此时玩家选择继续叫牌。通过 10^6 次蒙特卡罗采样并计算均值后知 $Q((13,2,\text{True}),1) \approx -0.023\ 305$，将其当作动作值的一个标准参考值。

现考虑用一般和加权重要性采样估计状态值，代码如下：

```
##【代码 3-7】增量式异策略评估算法

import numpy as np
import blackjack
from collections import defaultdict

'''
目标策略:如果点数小于 18,则继续叫牌,否则停牌
'''
def target_policy(state):
    player, dealer, ace = state
    return 0 if player >= 18 else 1      #0:停牌;1:要牌

'''
行为策略:均匀选择叫牌或停牌
'''
def behavior_policy(state):
    if np.random.rand() <= 0.5:
        return 0      #0:停牌
    else:
        return 1      #1:要牌

'''
增量式异策略每次访问蒙特卡罗策略评估:算法 3-7 的具体实现
'''
def offpolicy_firstvisit_mc_actionvalue(env,num_episodes=1000000):
    G_count = defaultdict(float)                    #记录状态-动作对的累积折扣奖励次数
```

```
    W_sum = defaultdict(float)                          #记录状态-动作对的累积重要性权重
    Q_bar_ord = defaultdict(float)                      #一般重要性采样动作值估计
    Q_bar_wei = defaultdict(float)                      #加权重要性采样动作值估计

    for i in range(num_episodes):
        #采集一条经验轨迹
        state = env.reset()                             #环境状态初始化
        one_mdp_seq = []                                #经验轨迹容器
        while True:
            action = behavior_policy(state)             #按行为策略选择动作
            next_state,reward,done,_ = env.step(action) #交互一步
            one_mdp_seq.append((state, action, reward)) #MDP序列
            if done:                                    #游戏是否结束
                break
            state = next_state

        #自后向前依次遍历MDP序列中的所有状态-动作对
        G = 0
        W = 1
        for j in range(len(one_mdp_seq)-1,-1,-1):
            sa_pair = (one_mdp_seq[j][0],one_mdp_seq[j][1])
            G = G+env.gamma*one_mdp_seq[j][2]           #累积折扣奖励
            W = W*(target_policy(sa_pair[0])/0.5)       #重要性权重
            if W == 0:                                  #如果权重为0,则退出本层循环
                break
            W_sum[sa_pair] += W                         #权重之和
            G_count[sa_pair] += 1                       #记录次数
                                                        #一般重要性采样估计
            Q_bar_ord[sa_pair] += (G-Q_bar_ord[sa_pair])/G_count[sa_pair]
                                                        #加权重要性采样估计
            Q_bar_wei[sa_pair] += (G-Q_bar_ord[sa_pair])*W/W_sum[sa_pair]

    return Q_bar_ord, Q_bar_wei

'''
主程序
'''
if __name__ == '__main__':
    env = blackjack.BlackjackEnv()                      #导入环境模型
    env.gamma = 1                                       #补充定义折扣系数

    Q_bar_ord,Q_bar_wei = offpolicy_firstvisit_mc_actionvalue(env)

    print('Ordinary action value of ((13,2,True),1) is {}'.
          format(Q_bar_ord[((13,2,True),1)]))
    print('Weighted action value of ((13,2,True),1) is {}'.
          format(Q_bar_wei[((13,2,True),1)]))
```

运行结果如下：

```
Ordinary action value of ((13,2,True),1) is -0.4722222222222224
Weighted action value of ((13,2,True),1) is 0.2438893461232294
```

可以看出，运行结果和参考值的出入还是比较大的。实际上，多次实验的结果表明，用异策略评估算法得到的值相当不稳定。这也从实验上说明了异策略算法理论完美但实用不足的缺陷。

【例 3-7】 用异策略蒙特卡罗强化学习求解 21 点游戏的最优策略，设行为策略为均匀选择停牌或要牌，代码如下：

```
##【代码 3-8】异策略蒙特卡罗强化学习算法代码

import numpy as np
import blackjack
from collections import defaultdict
import matplotlib.pyplot as plt

'''
异策略蒙特卡罗强化学习算法类
'''
class OffpolicyMCRL():
    ##类初始化
    def __init__(self,env,num_episodes=1000000):
        self.env = env
        self.nA = env.action_space.n                              #动作空间维度
        self.Q_bar = defaultdict(lambda: np.zeros(self.nA))       #动作值函数
        self.W_sum = defaultdict(lambda: np.zeros(self.nA))       #累积重要性权重
        self.t_policy = defaultdict(lambda: np.zeros(self.nA))    #目标策略
        self.b_policy = defaultdict(lambda: np.zeros(self.nA))    #行为策略
        self.num_episodes = num_episodes                          #最大抽样回合数

    ##初始化及更新目标策略
    def target_policy(self,state):
        if state not in self.t_policy.keys():
            player, dealer, ace = state
            action = 0 if player >= 18 else 1                     #0:停牌;1:要牌
        else:
            action = np.argmax(self.Q_bar[state])                 #最优动作值对应的动作
            self.t_policy[state] = np.eye(self.nA)[action]

        return self.t_policy[state]

    ##初始化行为策略
```

```
    def behavior_policy(self,state):
        self.b_policy[state] = [0.5,0.5]
        return self.b_policy[state]

    ##按照行为策略蒙特卡罗抽样产生一条经历完整的 MDP 序列
    def mc_sample(self):
        one_mdp_seq = []                                    #经验轨迹容器
        state = self.env.reset()                            #初始状态
        while True:
            action_prob = self.behavior_policy(state)
            action = np.random.choice(np.arange(len(action_prob)),
                                      p = action_prob)
            next_state,reward,done,_ = env.step(action)     #交互一步
            one_mdp_seq.append((state,action,reward))       #经验轨迹
            state = next_state
            if done:                                        #游戏是否结束
                break

        return one_mdp_seq

    ##基于值迭代的增量式异策略蒙特卡罗每次访问策略评估和改进
    def offpolicy_everyvisit_mc_valueiter(self,one_mdp_seq):
        #自后向前依次遍历 MDP 序列中的所有状态 - 动作对
        G = 0
        W = 1
        for j in range(len(one_mdp_seq) - 1, - 1, - 1):
            state = one_mdp_seq[j][0]
            action = one_mdp_seq[j][1]
            G = G + env.gamma * one_mdp_seq[j][2]           #累积折扣奖励
            W = W * (self.target_policy(state)[action]/self.behavior_policy(
                    state)[action])                         #重要性权重
            if W == 0:                                      #如果权重为 0,则退出本层循环
                break
            self.W_sum[state][action] += W                  #权重之和
            self.Q_bar[state][action] += (
                    G - self.Q_bar[state][action]) * W/self.W_sum[state][action]
            self.target_policy(state)                       #策略改进

    ##蒙特卡罗强化学习
    def mcrl(self):
        for i in range(self.num_episodes):
            one_mdp_seq = self.mc_sample()                  #抽样一条 MDP 序列
                                                            #蒙特卡罗策略评估和策略改进
            self.offpolicy_everyvisit_mc_valueiter(one_mdp_seq)

        return self.t_policy, self.Q_bar                    #输入策略和动作值
```

```
    ##绘制最优策略图像
    def draw(self,policy):
        true_hit = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == True and np.argmax(policy[x]) == 1]
        true_stick = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == True and np.argmax(policy[x]) == 0]
        false_hit = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == False and np.argmax(policy[x]) == 1]
        false_stick = [(x[1],x[0]) for x in policy.keys(
                ) if x[2] == False and np.argmax(policy[x]) == 0]

        plt.figure(1)
        plt.plot([x[0] for x in true_hit],
                 [x[1] for x in true_hit],'bo',label = 'HIT')
        plt.plot([x[0] for x in true_stick],
                 [x[1] for x in true_stick],'rx',label = 'STICK')
        plt.xlabel('dealer'), plt.ylabel('player')
        plt.legend(loc = 'upper right')
        plt.title('Usable Ace')
        filepath = 'code3-8 UsabelAce.png'
        plt.savefig(filepath, dpi = 300)

        plt.figure(2)
        plt.plot([x[0] for x in false_hit],
                 [x[1] for x in false_hit],'bo',label = 'HIT')
        plt.plot([x[0] for x in false_stick],
                 [x[1] for x in false_stick],'rx',label = 'STICK')
        plt.xlabel('dealer'), plt.ylabel('player')
        plt.legend(loc = 'upper right')
        plt.title('No Usable Ace')
        filepath = 'code3-8 NoUsabelAce.png'
        plt.savefig(filepath, dpi = 300)

'''
主程序
'''
if __name__ == '__main__':
    env = blackjack.BlackjackEnv()                    #导入环境模型
    env.gamma = 1                                     #补充定义折扣系数

    #定义方法
    agent = OffpolicyMCRL(env)
    #强化学习
    opt_policy,opt_Q = agent.mcrl()
    #打印结果
    for key in opt_policy.keys():
        print(key,": ",opt_policy[key],opt_Q[key])
    agent.draw(opt_policy)
```

运行结果如下：

```
#共 280 条数据,第 1 列为状态,第 2 列为策略,第 3 列为动作值
(20, 7, False) : [1. 0.]   [0.7739739  0.       ]
(21, 10, False) : [1. 0.] [0.88973242  0.        ]
⋮
⋮
(12, 6, True) : [0. 1.] [ - 1.            0.26530612]
(4, 1, False) : [1. 0.] [ - 0.74891775    - 1.       ]
```

由结果生成的策略如图 3-6 所示。可以看出，该策略和例 3-5 生成的策略相差还是比较大的，异策略算法在 21 点问题上表现并不好。

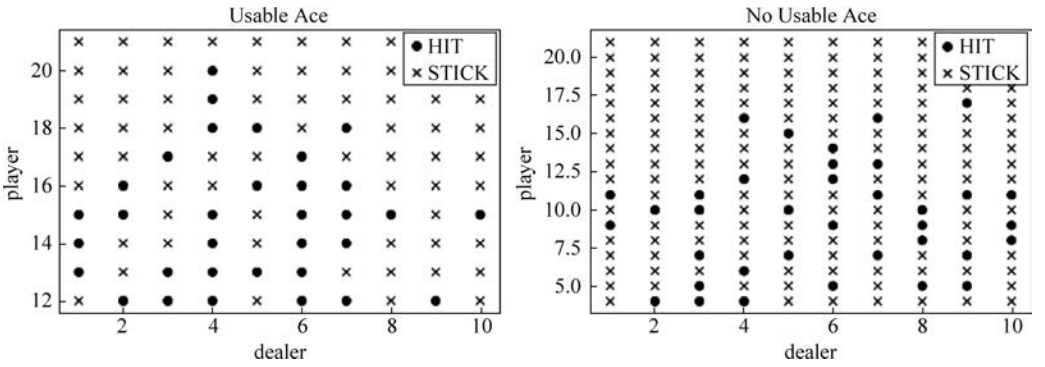


图 3-6　异策略蒙特卡罗强化学习算法求解 21 点问题策略示意图

3.5　蒙特卡罗树搜索

用蒙特卡罗法进行评估策略和改进的过程也称为蒙特卡罗搜索(Monte Carlo Search，MCS)。蒙特卡罗搜索的主要特点是在策略评估阶段保持策略不变，评估该策略下的所有动作值；在策略改进阶段用新的动作值改进所有状态下的策略。蒙特卡罗法搜索适用于状态空间较小的强化学习任务。当状态空间很大时，蒙特卡罗搜索就无能为力了，因为即使是随机生成了大量的经历完整的 MDP 序列，也很难覆盖所有的状态值。此时要使用蒙特卡罗树搜索(Monte Carlo Tree Search，MCTS)进行策略评估。

3.5.1　MCTS 的基本思想

MCTS 主要适用于基于零和博弈(输赢奖励之和为 0)的对战游戏，这类游戏往往具有非常明显的状态转移概率，但是状态空间尺度异常庞大，导致几乎不可能计算和保存全部动作值。以围棋游戏为例，在某一棋局下，棋手落子之后，棋局的转移是不言而喻的，但一副标

准围棋不同的棋局数为 $3^{361} \approx 10^{170}$ 个，要计算和保存每个棋局的最优策略显然是不现实的。

其实，棋手在对弈时只关心在当前棋局下如何落子，对大量其他无关棋局并不感兴趣。也就是说，在大规模离散状态空间强化学习任务中，为了避免计算和保存大量动作值和策略，可以只评估和改进当前状态下的动作值和策略。这种只聚焦于特定状态的策略评估和改进方法称为决策时间规划(Decision-time Planning)。

MCTS 就是一种典型的决策时间规划方法，其核心思想是：从当前状态出发，利用已有的策略和随机策略构建一棵搜索树，搜索树的根节点为当前状态，叶节点为终止状态，从根节点到叶节点的路径构成一条经历完整的 MDP 序列，对搜索树中的所有经历完整的 MDP 序列进行回溯，获得当前状态下的各动作值，最后依据这些动作值进行当前状态下的策略选择。同时，当前状态下的新策略被添加或更新到已有策略中(可能以表格或函数的方式表示)，以便在后续状态的策略评估中当成已有策略使用。

对 MCTS 基本思想的理解需要注意以下几点：

(1) MCS 维护的是整个动作值表和策略表，MCTS 只维护部分策略表(策略函数)，而且策略表会随着搜索的进行逐渐完善。MCTS 的策略选择不仅依据其所维护的部分策略表，也依据一些启发式随机策略。

(2) MCS 每局迭代都要更新整个动作值表和策略表，虽然并不是每个动作值都会被更新，但未被更新的动作值被看作更新值为 0。MCTS 每次迭代仅计算当前状态下的动作值，待策略选择完成以后，这些动作值就被删除了，并不保存，只有当前状态下的新策略会被添加或更新到已有策略中。

(3) MCTS 在生成搜索树时要使用两种策略：一是策略表(或策略函数)中已有的策略，这种策略称为树中策略；另一种是策略表(或策略函数)中没有的策略，这种策略称为树外策略。树外策略一般是启发式随机策略或准确度较低但决策时间短的快速策略。

(4) MCTS 擅长于基于零和博弈的对战游戏，特别是棋类游戏。DeepMind 公司开发的围棋程序 AlphaGo 和 AlphaGo Zero 就使用了 MCTS 进行决策，详细讨论见本书第 9 章。

3.5.2 MCTS 的算法流程

和 MCS 一样，MCTS 的算法流程也主要分为策略评估、策略选择和策略改进三部分。策略评估计算当前状态下的各动作值，又可以分为生成搜索树和回溯搜索树两部分。策略选择根据策略评估阶段得到的动作值，使用贪心策略选择当前状态下的动作。策略改进将当前状态下的改进策略加入或更新到已有策略中。

构建搜索树有深度优先和广度优先两种方案。深度优先逐条构建并评估从根节点到叶节点的完整 MDP 序列，广度优先则先构建整个搜索树，再进行回溯评估。显然，广度优先方案需要大量的空间来存储搜索树，空间复杂度过高，所以实际中更多使用深度优先方案。

以下结合一个简单的案例来讲解 MCTS 的算法流程。

【例 3-8】 数 21 游戏

数 21 游戏是一个简单而有趣的双人数数游戏。游戏双方(设为 A 和 B)从 1 开始轮流数数,每次只能数 1、2 或 3 个数,先数到 21 者为胜方。

显然,数 21 游戏的状态空间为 $\mathbf{S}=\{0,1,2,\cdots,21\}$,动作空间为 $\mathbf{A}=\{1,2,3\}$。从这一点上看,数 21 游戏的状态空间并不大,不属于 MCTS 所擅长的大规模离散状态空间强化学习任务,但一方面,本例是希望用一个简单的任务来说明 MCTS 的计算流程;另一方面,数字"21"其实是可以无限增大的,动作空间维度也可以增大,当增大到足够大时,这就是一个大规模离散状态空间强化学习任务了。

另外,数 21 游戏是有非常明确的最优策略的(先留给读者思考),而明确的最优策略有助于检验 MCTS 算法的有效性。

进一步介绍之前,先做以下假设:

(1) 假设参与者 A 作为智能体,参与者 B 和其他因素作为环境。

(2) 假设当前状态为 $s=10$,智能体 A 需要做出下一步数几个数的策略。

(3) 假设树中策略(已有策略)见表 3-1,表中"\"表示该状态尚无树中策略。

(4) 假设树外策略为从 1、2、3 中均匀随机选择一个数。

(5) 假设参与者 B(作为环境的一部分)的策略为从 1、2、3 中均匀随机选择一个数,参与者 B 的策略简称为环境策略。

(6) 若最终参与者 A 获胜,则获得奖励 1;若失败,则获得奖励 −1;未达终止状态的奖励均为 0。

表 3-1 树中策略(已有策略)

状 态	动 作	状 态	动 作	状 态	动 作
0	\	7	\	14	\
1	\	8	2	15	\
2	\	9	\	16	\
3	2	10	\	17	2
4	\	11	\	18	\
5	1	12	1	19	\
6	\	13	\	20	1

以下从深度优先生成搜索树、回溯搜索树、动作选择、策略改进 4 个步骤来阐述 MCTS 的算法流程。

1. 深度优先生成搜索树

从当前状态出发,选择一个动作,使用树中策略和树外策略,生成一条从当前状态到终止状态的完整 MDP 序列。

从 $s=10$ 出发，选择动作 $a=2$，生成的一条直到终止状态 $s_T=21$ 的完整 MDP 序列如图 3-7 所示，包括树外策略、树中策略、参与者 B 的随机策略（环境策略），箭头上的数字表示动作，即数数的个数。图 3-7 所示的 MDP 序列最后由智能体（玩家 A）数到了 21，所以对于该 MDP 序列来讲，智能体获得回报 1。

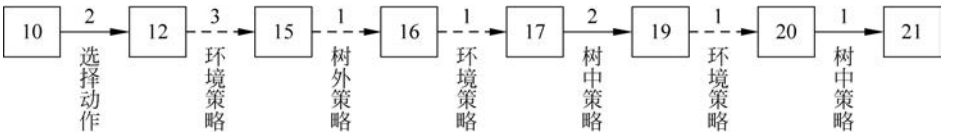


图 3-7　深度优先生成搜索树示意图

2. 回溯搜索树

按照深度优先生成搜索树的方法，为当前状态下的每种状态-动作对生成相同数量的完整 MDP 序列，然后回溯统计从每种状态-动作对往后的所有完整 MDP 序列的回报。

设为每种状态-动作对都生成了 10 条完整的 MDP 序列，其中从状态-动作对 $(s,a)=(10,1)$ 出发生成的 10 条 MDP 序列最终有 6 条回报为 -1，4 条回报为 $+1$，故对状态-动作对 $(s,a)=(10,1)$ 的动作值估计为 $Q((10,1))=-2$，同理可得 $Q((10,2))=6$，$Q((10,3))=-6$，如图 3-8 所示。

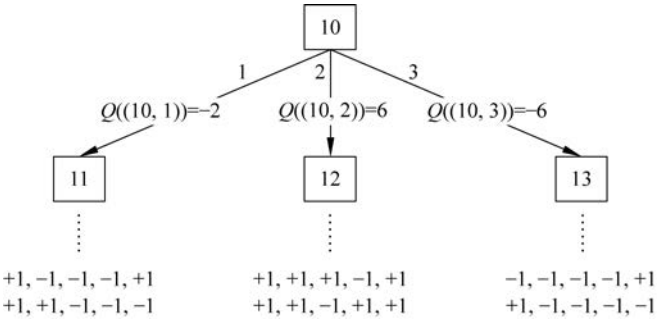


图 3-8　回溯搜索树示意图

3. 动作选择

根据回溯搜索树后得到的当前状态下的各动作值函数，应用贪婪策略进行动作选择。从图 3-8 回溯搜索树的结果可知，在 $s=10$ 时应选择的动作为 $a=2$，即数两个数。

4. 策略改进

将当前状态下的新策略加入或更新到策略表中，若原策略表中无当前状态下的策略，则直接添加；若已有当前状态下的策略，但和新策略不一样，则用新策略覆盖旧策略。

在本例中，表 3-1 中 $s=10$ 时无策略，所以直接将 $\pi(10)=2$ 加入新策略表中，改进后的策略见表 3-2。

表 3-2 改进后的树中策略

状 态	动 作	状 态	动 作	状 态	动 作
0	\	7	\	14	\
1	\	8	2	15	\
2	\	9	\	16	\
3	2	10	2	17	2
4	\	11	\	18	\
5	1	12	1	19	\
6	\	13	\	20	1

理解 MCTS 的算法流程还应该注意以下几点：

(1) 在实际应用中，因为状态空间很大，所以树中策略一般是以策略函数而非表格形式表达的，此时更新策略的过程实际上就是训练策略函数。

(2) 回溯搜索树完成后，搜索树的数据就被丢弃了，不需要保存。

(3) 在实际应用中，由于完整搜索树可能非常庞大，所以只能生成和回溯搜索树的一部分，用部分采样的结果来近似整个搜索树。

(4) 在实际应用中，当前状态下的某些动作是完全没有意义的，所以在生成搜索树时可以忽略这些动作。

3.5.3 基于 MCTS 的强化学习算法

在 MCTS 的基础上，可以设计基于 MCTS 的强化学习算法，算法流程如下：

算法 3-9 基于蒙特卡罗树搜索的强化学习算法

1. 输入：环境模型 MDP($\mathbf{S},\mathbf{A},R,\gamma$)，树中策略 π_0，树外策略 π'，迭代局数 num_episodes
2. 初始化：初始化树中策略 $\pi=\pi_0$
3. 过程：
4. 　　循环：ep=1～num_episodes
5. 　　环境状态初始化：s
6. 　　循环：直到到达终止状态
7. 　　根据 MCTS 选择动作：a
8. 　　执行一次交互：$s'\leftarrow s,a$
9. 　　策略改进：将 $\pi(s)=a$ 添加或改进到树中策略 π 中
10. 　　状态更新：$s\leftarrow s'$
11. 输出：最终策略 π

3.5.4 案例和代码

本节继续以数 21 游戏为例，首先给出游戏的环境模型代码，然后给出用 MCTS 进行一

次动作选择的代码，即例 3-8，最后给出用算法 3-9 求解数 21 游戏最优策略的代码。

数 21 游戏的环境模型代码如下：

```
##【代码 3-9】数 21 游戏环境模型代码

import random

class Count21Env():
    def __init__(self,num_max = 21,num_least = 1,num_most = 3,gamma = 1):
        self.num_max = num_max                              #数数的终点，也是终止状态
        self.num_least = num_least                          #每次最少数的数
        self.num_most = num_most                            #每次最多数的数
        self.start = 0                                      #数数的起点，也是初始状态
        self.goal = num_max                                 #终止状态
        self.state = None                                   #当前状态
        self.gamma = gamma                                  #折扣系数

        self.sspace_size = self.num_max + 1                 #状态个数
        self.aspace_size = self.num_most - self.num_least + 1  #动作个数

    ##获取状态空间
    def get_sspace(self):
        return [i for i in range(self.start, self.num_max + 1)]

    ##获取动作空间
    def get_aspace(self):
        return [i for i in range(self.num_least, self.num_most + 1)]

    ##环境初始化
    def reset(self):
        self.state = self.start
        return self.state

    #庄家策略，作为环境的一部分，随机选择一个动作
    def get_dealer_action(self):
        return random.choice(self.get_aspace())

    #进行一个时间步的交互
    def step(self,action):
        self.state += action                                #玩家数 action 个数

        if self.state > self.goal:                          #超过终止状态，庄家获胜
            reward = -1                                     #庄家获胜，玩家得 -1 分
            end = True
            info = "Player count then lose"
```

```
        elif self.state == self.goal:                    #到达终止状态,玩家获胜
            reward = 1                                   #玩家获胜,玩家得 1 分
            end = True
            info = "Player count then win"
        else:                                            #庄家继续数数
            self.state += self.get_dealer_action()       #庄家数数
            if self.state > self.goal:                   #超过终止状态,玩家获胜
                reward = 1                               #玩家获胜,玩家得 1 分
                end = True
                info = "Dealer count then lose"
            elif self.state == self.goal:
                reward = -1                              #庄家获胜,玩家得 -1 分
                end = True
                info = "Dealer count then win"
            else:
                reward = 0                               #游戏继续,玩家得 0 分
                end = False
                info = "Keep Going"

        return self.state,reward,end,info
```

将此代码保存为 Count21.py,并和本节后续代码放在同一个文件夹中,以备调用。假设当前状态为 $s=10$,使用 MCTS 在当前状态下进行一次策略选择的代码如下:

```
##【代码 3-10】基于 MCTS 的单次动作选择代码

import numpy as np
import random

##树外策略
def offtree_policy(env):
    return random.choice(env.get_aspace())        #随机选择一个动作

##树中策略
def create_ontree_policy(env):                    #如表 3-1 所示
    ontree_policy = {}
    for state in env.get_sspace():
        ontree_policy[state] = None
    ontree_policy[3] = 2
    ontree_policy[5] = 1
    ontree_policy[8] = 2
    ontree_policy[12] = 1
    ontree_policy[17] = 2
    ontree_policy[20] = 1

    return ontree_policy
```

```
#蒙特卡罗树搜索进行一次策略选择
def mcts(env,state_cur,ontree_policy,num_mdpseq=100):
    Q = {}                                  #当前状态下的各动作值容器
    for action_ in env.get_aspace():        #遍历当前状态下的各动作
        Q[action_] = 0
        for i in range(num_mdpseq):         #每种状态-动作对生成相同数目的完整 MDP 序列
            env.state = state_cur
            action = action_
            while True:                     #生成搜索树
                state,reward,done,info = env.step(action)
                #如果到达终止状态
                if done:
                    Q[action_] += reward    #回溯搜索树
                    break

                #根据树外或树中策略选择动作
                if ontree_policy[state] == None:
                    action = offtree_policy(env)
                else:
                    action = ontree_policy[state]

    action_opt = np.argmax(np.array([x for x in Q.values()])) + 1

    return Q,action_opt

#main function
if __name__ == '__main__':
    import Count21
    env = Count21.Count21Env()

    ontree_policy = create_ontree_policy(env)
    num_mdpseq = 100000
    state_cur = 10
    Q, action_opt = mcts(env,state_cur,ontree_policy,num_mdpseq)

    print(Q,action_opt)
```

运行结果如下：

```
{1: 37098, 2: 36314, 3: 34774} 1
```

也就是说，根据现有的树中和树外策略，由蒙特卡罗树搜索对每种状态-动作对进行100 000次模拟后得到$\pi(10)=1$，即参与者A数1个数。

最后，用基于MCTS的强化学习算法求解数21游戏的完整代码如下：

```
##【代码 3-11】基于 MCTS 强化学习算法求解数 21 游戏的最优策略

import numpy as np
import random

"""
基于 MCTS 的强化学习类
"""
class MCTS_RL():
    def __init__(self,env,episode_max = 100,num_mdpseq = 100):
        self.env = env
        self.episode_max = episode_max
        self.num_mdpseq = num_mdpseq

        self.ontree_policy = self.create_ontree_policy()

    ##创建初始树中策略
    def create_ontree_policy(self):
        ontree_policy = {}                          #用字典表示策略
        for state in env.get_sspace():              #遍历每种状态
            ontree_policy[state] = None             #初始化为无策略

        return ontree_policy                        #返回策略

    ##添加或改进策略到树中策略
    def update_ontree_policy(self,state,action):
        if self.ontree_policy[state] == None:
            self.ontree_policy[state] = action
        else:
            self.ontree_policy[state] = action

    ##树外策略,随机选择一个动作
    def offtree_policy(self):
        return random.choice(self.env.get_aspace())

    ##蒙特卡罗树搜索进行一次策略选择
    def mcts(self,state_cur):
        Q = {} #当前状态下的各动作值容器
        for action_ in self.env.get_aspace():       #遍历当前状态下的各动作
            Q[action_] = 0
            for i in range(self.num_mdpseq):        #生成搜索树
                self.env.state = state_cur
                action = action_
                while True:
                    state,reward,done,info = self.env.step(action)
                    #如果到达终止状态
```

```
                    if done:
                        Q[action_] += reward                    #回溯搜索树
                        break

                    #根据树外或树中策略选择动作
                    if self.ontree_policy[state] == None:
                        action = self.offtree_policy()
                    else:
                        action = self.ontree_policy[state]

        action_opt = np.argmax(np.array([x for x in Q.values()])) + 1

        return action_opt

    ##基于MCTS的强化学习
    def mcts_rl(self):
        for i in range(self.episode_max):
            state_cur = self.env.reset()                        #状态初始化
            while True:
                action = self.mcts(state_cur)                   #MCTS动作选择
                self.update_ontree_policy(state_cur,action)     #策略改进
                self.env.state = state_cur
                state_cur,reward,done,info = self.env.step(action)
                if done:
                    break

        return self.ontree_policy

"""
主函数
"""
if __name__ == '__main__':
    import Count21
    env = Count21.Count21Env()

    episode_max = 100
    num_mdpseq = 1000
    agent = MCTS_RL(env,episode_max,num_mdpseq)

    ontree_policy = agent.mcts_rl()
    print(ontree_policy)
```

运行结果如下：

```
{0: 1, 1: None, 2: 3, 3: 2, 4: 1, 5: 3, 6: 3, 7: 2, 8: 1, 9: 2, 10: 3, 11: 2, 12: 1, 13: 2, 14: 3,
15: 2, 16: 1, 17: 1, 18: 3, 19: 2, 20: 1, 21: None}
```

根据运行结果得出的策略见表 3-3。

表 3-3 数 21 游戏最终策略(也是最优策略)

状 态	动 作	状 态	动 作	状 态	动 作
0	1	7	2	14	3
1	\	8	1	15	2
2	3	9	2	16	1
3	2	10	3	17	1
4	1	11	2	18	3
5	3	12	1	19	2
6	3	13	2	20	1

其实,表 3-3 所示的策略就是数 21 游戏的最优策略。参与者 A(作为智能体)在倒数第 2 轮只要能够数到 17,则不论参与者 B(作为环境的一部分)数 1、2、3 中的哪一个数,参与者 A 都能够在最后一轮数到 21,从而获胜。如此反推,参与者 A 应该要尽量数到 17、13、9、5、1 这几个数,表 3-3 中的策略正是这样。这也是为什么参与者 A 在 1 时没有策略的原因,因为如果参与者 A 先数,按照最优策略,参与者 A 首先一定数 1,接下来一定是参与者 B 数,故参与者 A 永远不会从 1 接着数。