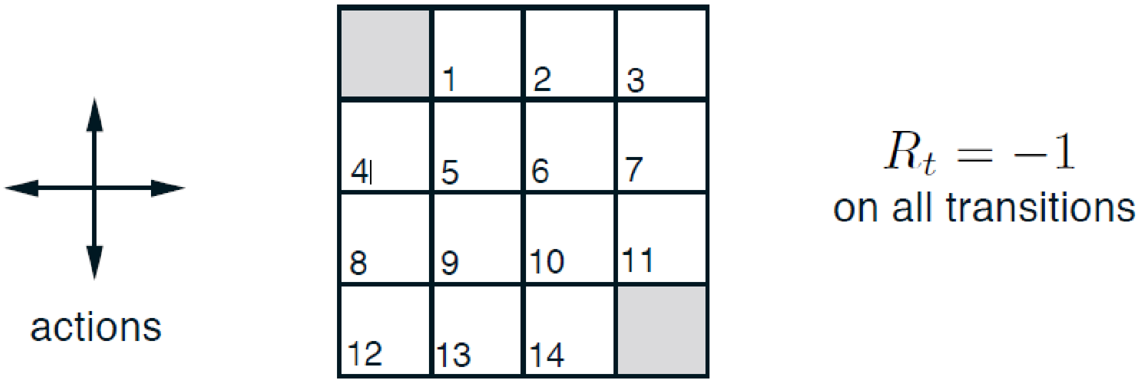
强化学习基础篇（二十二）DP小型网格问题

该问题基于《Reinforcement Learning: An Introduction》在第四章的例4.1。

1、问题描述

考虑下面的这个4*4的网格图



非终止状态集合$S = 1，2, \ldots, 14$。每个状态有四种可能的动作，$A = up, down, left, right$。每个动作会导致状态转移，但当动作会导致智能体移出网格时，状态保持不变。比如，$p(6, -1 \mid 5, right) = 1$，$p(7, -1 \mid 7, right) = 1$和对于任意$r \in R$，都有$p(10, r \mid 5, right) = 0$。这是一个无折扣的分幕式任务。在到达终止状态之前，所有动作的收益均为-1。终止状态在图中以阴影显示（尽管图中显示了两个格子，但实际仅有一个终止状态）。对于所有的状态s，s'以及动作a，期望的收益函数均为$r(s, a, s') = -1$。假设智能体采取等概率随机策略（所有动作等可能执行），我们需要计算在迭代策略评估中价值函数序列的收敛情况。

2、实现过程

2.1、环境定义

首先导入库函数以及定义环境信息：

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.table import Table

# 定义网格世界大小
WORLD_SIZE = 4

# 把动作定义为对x，y坐标的增减改变
# left, up, right, down
ACTIONS = [np.array([0, -1]),  # 向上
           np.array([-1, 0]),  # 向左
           np.array([0, 1]),   # 向下
           np.array([1, 0])]   # 向右
# 该问题中每个动作选择的概率为0.25
ACTION_PROB = 0.25
# 定义画图会用到的动作
ACTIONS_FIGS=[ '←', '↑', '→', '↓']
```

然后定义左上角和右下角两个坐标（0,0）与（WORLD_SIZE - 1，WORLD_SIZE - 1）两个点为terminal

```python
def is_terminal(state):
    '''
    返回是否为terminal
    '''
    x, y = state
    return (x == 0 and y == 0) or (x == WORLD_SIZE - 1 and y == WORLD_SIZE -
1)
```

2.2、定义动作执行过程

```python
def step(state, action):
    # 当到达terminal时，下一步状态不变，奖励为0
    if is_terminal(state):
        return state, 0
    # 计算下一个状态
    next_state = (np.array(state) + action).tolist()
    x, y = next_state
    # 当运动未知超出格子世界则在原位置不变
    if x < 0 or x >= WORLD_SIZE or y < 0 or y >= WORLD_SIZE:
        next_state = state

    reward = -1
    return next_state, reward
```

2.3、辅助函数

以下辅助函数主要用于画图

```python
def draw_image(image):
    fig, ax = plt.subplots()
    ax.set_axis_off()
    tb = Table(ax, bbox=[0, 0, 1, 1])

    nrows, ncols = image.shape
    width, height = 1.0 / ncols, 1.0 / nrows

    # Add cells
    for (i, j), val in np.ndenumerate(image):
        tb.add_cell(i, j, width, height, text=val,
                    loc='center', facecolor='white')

        # Row and column labels...
    for i in range(len(image)):
        tb.add_cell(i, -1, width, height, text=i+1, loc='right',
                    edgecolor='none', facecolor='none')
        tb.add_cell(-1, i, width, height/2, text=i+1, loc='center',
                    edgecolor='none', facecolor='none')
    ax.add_table(tb)
```

以下辅助函数用户策略描述：

```python
def draw_policy(optimal_values):
    fig, ax = plt.subplots()
```

```python
    ax.set_axis_off()
    tb = Table(ax, bbox=[0, 0, 1, 1])

    nrows, ncols = optimal_values.shape
    width, height = 1.0 / ncols, 1.0 / nrows

    # Add cells
    for (i, j), val in np.ndenumerate(optimal_values):
        next_vals=[]
        for action in ACTIONS:
            next_state, _ = step([i, j], action)
            next_vals.append(optimal_values[next_state[0],next_state[1]])

        best_actions=np.where(next_vals == np.max(next_vals))[0]
        val=''
        for ba in best_actions:
            val+=ACTIONS_FIGS[ba]

        # add state labels
        if [i, j] == [0,0]:
            val = "terminal"
        if [i, j] == [WORLD_SIZE - 1,WORLD_SIZE - 1]:
            val = "terminal"

        tb.add_cell(i, j, width, height, text=val,
                loc='center', facecolor='white')

    # Row and column labels...
    for i in range(len(optimal_values)):
        tb.add_cell(i, -1, width, height, text=i+1, loc='right',
                    edgecolor='none', facecolor='none')
        tb.add_cell(-1, i, width, height/2, text=i+1, loc='center',
                   edgecolor='none', facecolor='none')

    ax.add_table(tb)
```

2.4、使用迭代策略评估算法估算状态值函数

使用迭代策略评估算法估算状态值函数，遵循的算法如下：

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

这里同时考虑了是否使用In-Place动态规划（In-place dynamic programming）。

在基于同步动态规划的值迭代算法中，存储了两个值函数的备份，分别是$v_{new}(s)$和$v_{old}(s)$。

$$v_{new}(s)=\max_a(r+\gamma\sum_{s'\in S}p(s'|s,a)v_{old}(s'))$$

即在计算过程中，通过赋值的方式使旧的状态值作为下一次计算新的状态值。
而In-place动态规划（In-Place Dynamic Programming，IPDP）则是去掉旧的状态值$v_{old}(s)$，只保留最新的状态值$v_{new}(s)$，在更新的过程中可以减少存储空间的浪费。

$$v(s)=\max_a(r+\gamma\sum_{s'\in S}p(s'|s,a)v(s'))$$

直接原地更新下一个状态值$v(s)$，而不像同步迭代那样需要额外存储新的状态值$v_{new}(s)$。在这种情况下，按何种次序更新状态值有时候会更具有意义。

```python
def compute_state_value(in_place=True, discount=1.0):
    # 初始化状态值函数为0
    new_state_values = np.zeros((WORLD_SIZE, WORLD_SIZE))
    # 在中间几个迭代进行可视化绘图
    draw_iteration=[0,1,2,3,10]
    iteration = 0
    while True:
        if iteration in draw_iteration:
            draw_image(np.round(new_state_values, decimals=2))
        # 判断是否使用In-Place动态规划（In-place dynamic programming）
        if in_place:
            state_values = new_state_values
        else:
            state_values = new_state_values.copy()
        old_state_values = state_values.copy()

        # 遍历所有状态
        for i in range(WORLD_SIZE):
            for j in range(WORLD_SIZE):
                value = 0
                # 遍历所有动作,按DP算法更新
                for action in ACTIONS:
                    (next_i, next_j), reward = step([i, j], action)
                    value += ACTION_PROB * (reward + discount * state_values[next_i, next_j])
                new_state_values[i, j] = value

        # 误差小于门限则停止更新
        max_delta_value = abs(old_state_values - new_state_values).max()
        if max_delta_value < 1e-4:
            draw_image(np.round(new_state_values, decimals=2))
            break

        iteration += 1

    return new_state_values, iteration
```

3、实验结果

运行如下代码测试在使用In-place动态规划（In-Place Dynamic Programming，IPDP）的结果：

```python
_, asycn_iteration = compute_state_value(in_place=True)
```

结果整理后如下：

```
In-place: 113 iterations
```

K=0

	1	2	3	4
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

	1	2	3	4
1	terminal	←↑→↓	←↑→↓	←↑→↓
2	←↑→↓	←↑→↓	←↑→↓	←↑→↓
3	←↑→↓	←↑→↓	←↑→↓	←↑→↓
4	←↑→↓	←↑→↓	←↑→↓	terminal

K=1

	1	2	3	4
1	0.0	-1.0	-1.25	-1.31
2	-1.0	-1.5	-1.69	-1.75
3	-1.25	-1.69	-1.84	-1.9
4	-1.31	-1.75	-1.9	0.0

	1	2	3	4
1	terminal	←	←	←
2	↑	←↑	↑	↑
3	↑	←	←↑	↓
4	↑	←	→	terminal

K=2

	1	2	3	4
1	0.0	-1.94	-2.55	-2.73
2	-1.94	-2.81	-3.24	-3.4
3	-2.55	-3.24	-3.57	-3.22
4	-2.73	-3.4	-3.22	0.0

	1	2	3	4
1	terminal	←	←	←
2	↑	←↑	↑	↑
3	↑	←	→↓	↓
4	↑	←	→	terminal

K=3

	1	2	3	4
1	0.0	-2.82	-3.83	-4.18
2	-2.82	-4.03	-4.71	-4.88
3	-3.83	-4.71	-4.96	-4.26
4	-4.18	-4.88	-4.26	0.0

	1	2	3	4
1	terminal	←	←	←
2	↑	←↑	↑	↑
3	↑	←	→↓	↓
4	↑	←	→	terminal

K=10

	1	2	3	4
1	0.0	-7.83	-11.12	-12.23
2	-7.83	-10.42	-11.77	-11.86
3	-11.12	-11.77	-11.05	-8.81
4	-12.23	-11.86	-8.81	0.0

	1	2	3	4
1	terminal	←	←	←
2	↑	←↑	←	↓
3	↑	↑	→↓	↓
4	↑	→	→	terminal

K=113

	1	2	3	4
1	0.0	-14.0	-20.0	-22.0
2	-14.0	-18.0	-20.0	-20.0
3	-20.0	-20.0	-18.0	-14.0
4	-22.0	-20.0	-14.0	0.0

	1	2	3	4
1	terminal	←	←	←↓
2	↑	←↑	←↓	↓
3	↑	↑→	→↓	↓
4	↑→	→	→	terminal

运行如下代码测试在不使用In-place动态规划（In-Place Dynamic Programming，IPDP）的结果：

```
values, sync_iteration = compute_state_value(in_place=False)
```

结果整理后如下：

```
Synchronous: 172 iterations
```

	1	2	3	4
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0

	1	2	3	4
1	terminal	←↑→↓	←↑→↓	←↑→↓
2	←↑→↓	←↑→↓	←↑→↓	←↑→↓
3	←↑→↓	←↑→↓	←↑→↓	←↑→↓
4	←↑→↓	←↑→↓	←↑→↓	terminal

K=0

	1	2	3	4
1	0.0	-1.0	-1.0	-1.0
2	-1.0	-1.0	-1.0	-1.0
3	-1.0	-1.0	-1.0	-1.0
4	-1.0	-1.0	-1.0	0.0

	1	2	3	4
1	terminal	←	←↑→↓	←↑→↓
2	↑	←↑→↓	←↑→↓	←↑→↓
3	←↑→↓	←↑→↓	←↑→↓	↓
4	←↑→↓	←↑→↓	→	terminal

K=1

	1	2	3	4
1	0.0	-1.75	-2.0	-2.0
2	-1.75	-2.0	-2.0	-2.0
3	-2.0	-2.0	-2.0	-1.75
4	-2.0	-2.0	-1.75	0.0

	1	2	3	4
1	terminal	←	←	←↑→↓
2	↑	←↑	←↑→↓	↓
3	↑	←↑→↓	→↓	↓
4	←↑→↓	→	→	terminal

K=2

K=3

	1	2	3	4
1	0.0	-2.44	-2.94	-3.0
2	-2.44	-2.88	-3.0	-2.94
3	-2.94	-3.0	-2.88	-2.44
4	-3.0	-2.94	-2.44	0.0

	1	2	3	4
1	terminal	←	←	←↓
2	↑	←↑	←↓	↓
3	↑	↑→	→↓	↓
4	↑→	→	→	terminal

K=10

	1	2	3	4
1	0.0	-6.14	-8.35	-8.97
2	-6.14	-7.74	-8.43	-8.35
3	-8.35	-8.43	-7.74	-6.14
4	-8.97	-8.35	-6.14	0.0

	1	2	3	4
1	terminal	←	←	←↓
2	↑	←↑	←↓	↓
3	↑	↑→	→↓	↓
4	↑→	→	→	terminal

K=172

	1	2	3	4
1	0.0	-14.0	-20.0	-22.0
2	-14.0	-18.0	-20.0	-20.0
3	-20.0	-20.0	-18.0	-14.0
4	-22.0	-20.0	-14.0	0.0

	1	2	3	4
1	terminal	←	←	←↓
2	↑	←↑	←↓	↓
3	↑	↑→	→↓	↓
4	↑→	→	→	terminal