因为在射击问题中，有以下几个特点：

- 是一个单向选择和转移的过程，最后形成了一张有向无环图。这就决定了我们可以从后向前一步步**回溯**，自下向上逐层计算状态价值函数 $v_\pi(s)$ 和动作价值函数 $q_\pi(s,a)$，直到开始状态。其计算过程是：$v_T \rightarrow q_\pi \rightarrow v_\pi \rightarrow q_\pi \rightarrow v_\pi$，如图 mdp-8 所示。
- 每个分支的细节分析起来比较多，但实际上分支之间的区别只存在于状态转移的过程的具体概率数值上。比如图 mdp-7 所示，其中的"红3"动作后的状态转移概率是 [0.25,0.7.0.05]，而动作"红5"的状态转移概率是 [0.2,0.75,0.05]。这种细节只是导致计算结果不同，但概念是相同的。
- 而另外一个重要的策略选择问题，我们暂时用"红:蓝=0.4:0.6"的统计结果设定了，在两次射击时都是如此。但是在实际问题中，游客各有各的策略，这种统一的设定只是一种统计结果。游乐场老板虽然精明，但是游客们也不是傻子，谁都知道连续两次选择红色气球是有机会得到最高值 6 分的奖励的，只不过是能不能顺利实现的问题。

迭代问题

在前面的射击气球问题中，我们根据贝尔曼期望方程中的状态价值函数 v_π 和动作价值函数 q_π 的定义，用反推的方法，手工计算出了各个节点的价值函数，以加深对价值函数定义的理解。

但是，如果有些问题没有定义终止状态的话，我们该从何下手来计算呢?

其实在学习马尔可夫奖励过程和贝尔曼方程时，曾经使用迭代方法来得到状态价值函数 v 的收敛值，而在这里可以使用同样的方法来解决没有终止状态的贝尔曼期望方程的问题。

穿越虫洞问题

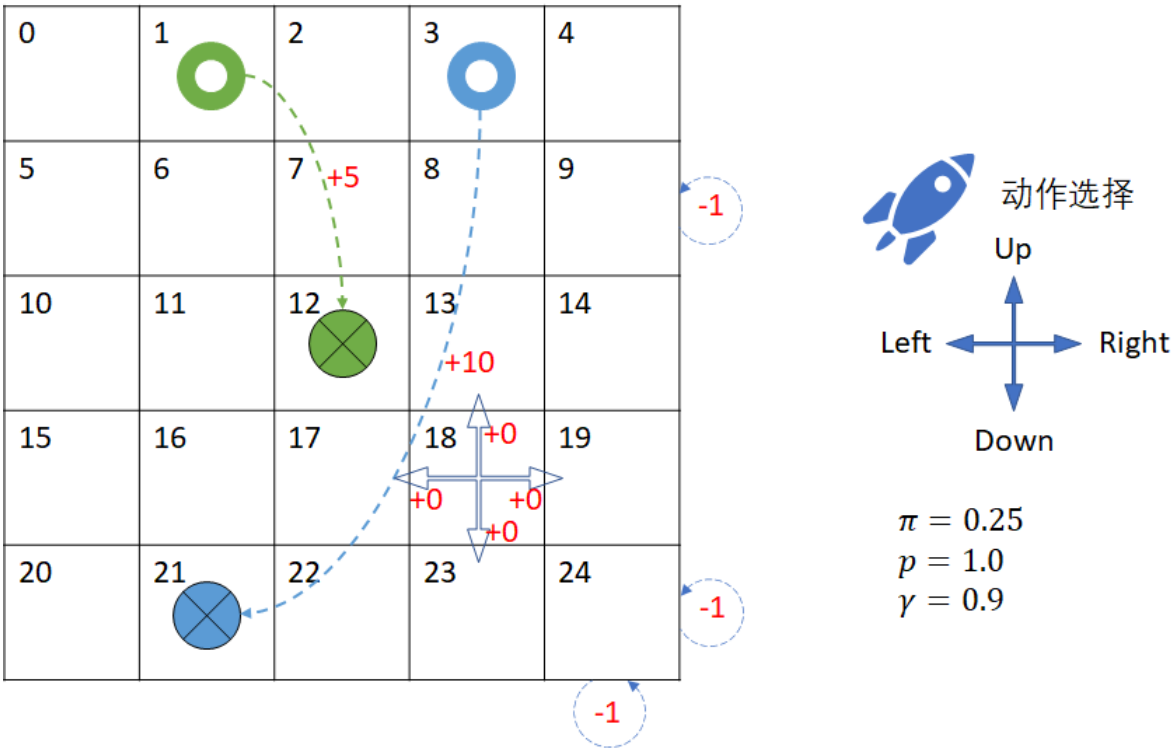

图 1

问题描述

- 在一个 5x5 的宇宙空间中，一艘探索太空的宇宙飞船可以任意向四个方向行驶，策略 $\pi = 0.25$。
- 比如在 s_{18} 处，如果选择向右行驶，将会以 $p = 1.0$ 的转移概率达到 19，并得到 0 的奖励。

- 如果在如 s_{24} 所示的角落处向右行驶或向下行驶，将会碰撞能量屏障使飞船受损，飞船位置不发生改变，得到 -1 的“奖励”，但并非终止状态。其它角落处也是如此，一共有 4 个角落状态（序号为：0，4，20，24）。
- 如果在如 s_9 所示的边界处向右行驶，将会碰撞能量屏障使飞船受损，得到 -1 的“奖励”，飞船位置不发生改变，但并非终止状态，还可以进一步行驶。其它边界处也是如此，一共有 12 个边界状态（序号为：1，2，3，5，9，10，14，15，19，21，22，23）。
- 在 s_1 和 s_3 处有两个虫洞：
    - 在 s_1 处进行下一步行驶时，无论任何方向，将无条件地达到 s_{12} 处，并得到 +5 的奖励，但后者并非终止状态。
    - 在 s_3 处进行下一步行驶时，无论任何方向，将无条件地达到 s_{21} 处，并得到 +10 的奖励，但后者并非终止状态。

    所以序号为 1,3 的两个状态可以从边界状态中去掉。

注意几点：

1. 没有终止状态，也就是说没有分幕，飞船可以一直行驶。
2. 到达 s_1, s_3 时，不是被立刻吸入虫洞，而是要进行下一步动作时才会时空转移。也就说在 s_1, s_3 并没有机会向上行驶而出界。
3. 关于边角位置的状态，如图 x 所示，以状态 s_2 为例：
    - 如果从该状态以 0.25 的概率选择向上移动，会以概率 1.0 回到 s_2，并有 -1 的奖励。所以说，这里面既有策略 π，又有转移概率 p，只不过只有一个下游状态，没有分支。
    - 如果从 s_2 以 0.25 的概率选择向下移动，会以 1.0 的概率转移到 s_7，得到 0 的奖励。

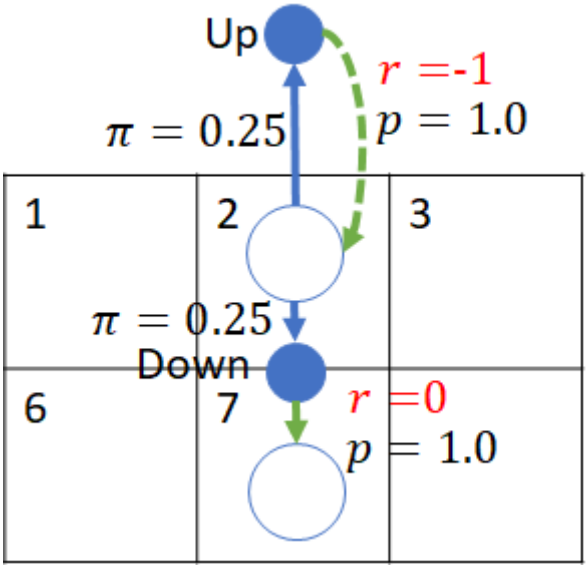

图 6

没有终点状态的话，我们无法确定任意一个状态的价值函数，进而算出其上游的动作价值函数。所以，我们必须研究一下马尔可夫决策过程下的贝尔曼期望方程的迭代解法了。

二级回溯

从前面的推导中，可以在已知 $q_\pi(s,a)$ 时计算出 $v_\pi(s)$（式5），也可以已知 $v_\pi(s)$ 计算出 $q_\pi(s,a)$（式2），这似乎变成了一个鸡生蛋蛋生鸡的死循环问题。能不能像贝尔曼方程那样，从$v_\pi(s')$ 计算出 $v_\pi(s)$ 来，从$q_\pi(s',a')$ 计算出 $q_\pi(s,a)$ 来呢？这样的话，我们就可以继续利用前面学过的迭代法或者矩阵法来方便地解出状态价值函数和动作价值函数了。

先绘制出两张二级回溯图，方便公式推导。如图 6 所示。

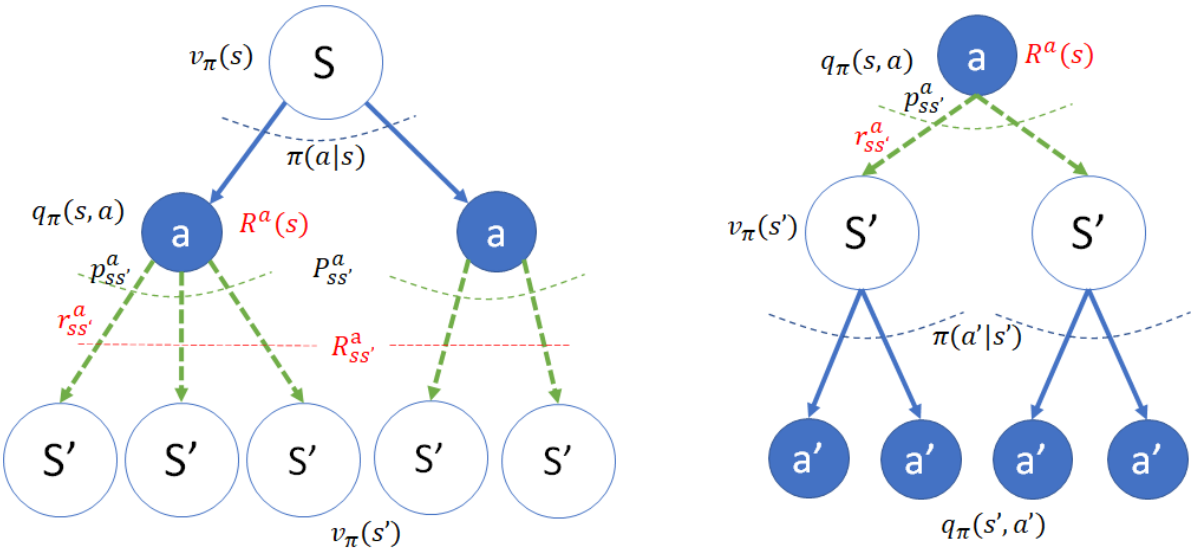

图 6

先看左图，我们的目的是想从 $v_\pi(s')$ 得到 $v_\pi(s)$，中间隔着一个 $q_\pi(s,a)$。如果把 $q_\pi(s,a)$ 的表达式带入 $v_\pi(s)$ 的表达式，就可以达到消去 $q_\pi(s,a)$ 的目的了。

于是可以把式 2.1 的 $q_\pi(s,a)$ 带入式 5：

$$
\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) q_\pi(s,a) && (式5) \\
(带入式2.1替换q_\pi \rightarrow) &= \sum_a \pi(a|s) \Big(\sum_{s'} p^a_{ss'} [r^a_{ss'} + \gamma v_\pi(s')] \Big) && (10.1) \\
(矩阵形式 \rightarrow) &= \sum_a \pi(a|s) \Big(P^a_{ss'} [R^a_{ss'} + \gamma V_\pi(s')] \Big) && (10.2) \\
(动作奖励函数形式 \rightarrow) &= \sum_a \pi(a|s) \Big(R^a(s) + \gamma P^a_{ss'} V_\pi(s') \Big) && (10.3)
\end{aligned}
\tag{10}
$$

式 10 就是 v_π 的迭代表达式。

再看右图，想把$v_\pi(s')$ 消掉，需要先得到它的表达式。

从式 5 把 s, a 换成 s', a'，可以得到式 11：

$$
v_\pi(s') = \sum_{a'} \pi(a'|s') q_\pi(s',a') \tag{11}
$$

把式 11 带入式 2：

$$
\begin{aligned}
q_\pi(s,a) &= \sum_{s'} p^a_{ss'} \left[r^a_{ss'} + \gamma v_\pi(s') \right] && (式2.1) \\
(带入式11替换v_\pi \rightarrow) &= \sum_{s'} p^a_{ss'} \Big(r^a_{ss'} + \gamma \sum_{a'} [\pi(a'|s') q_\pi(s',a')] \Big) && (12.1) \\
(动作奖励函数形式 \rightarrow) &= R^a(s) + \gamma \sum_{s'} p^a_{ss'} \Big(\sum_{a'} [\pi(a'|s') q_\pi(s',a')] \Big) && (12.2)
\end{aligned}
\tag{12}
$$

式 12 就是 q_π 的迭代的表达形式。

本节的公式有些多，下面总结一下，便于读者以后速查。

表 3 v_π, q_π 的互换速查表

=	v_π	q_π
v_π	$v_\pi(s)=\sum_a \pi(a \mid s)\left(\sum_{s'} p_{ss'}^a[r_{ss'}^a+\gamma v_\pi(s')]\right)$ (式10.1)	$v_\pi(s)=\sum_a \pi(a \mid s) q_\pi(s,a)$ (式5)
q_π	$q_\pi(s,a)=\sum_{s'} p_{ss'}^a[r_{ss'}^a+\gamma v_\pi(s')]$ (式2.1)	$q_\pi(s,a)=\sum_{s'} p_{ss'}^a\left(r_{ss'}^a+\gamma \sum_{a'}[\pi(a' \mid s') q_\pi(s',a')]\right)$ (式12.1)

在表 3 中，我们只给出了公式的原始形式，读者可以在实际用使用矩阵形式或是奖励函数形式。

建模

在强化学习中，经常会用图 x 这种方格（或长方格）来研究各种算法，所以有必要建立一个通用的模型，用数据定义模型的各种行为。

模型可以分为四个小部分

状态部分

```
# 状态空间 = 空间宽度 x 空间高度
GridWidth, GridHeight = 5, 5
# 起点，可以多个
StartStates = []
# 终点，可以多个
EndStates = []
```

- 空间宽度和高度可以不相等，比如 3 X 4。
- 有些场景需要定义起点，比如迷宫游戏。
- 很多场景需要定义终止状态，到达此状态后算是分幕结束。

动作部分

```
# 动作空间
LEFT, UP, RIGHT, DOWN  = 0, 1, 2, 3
Actions = [LEFT, UP, RIGHT, DOWN]
# 初始策略
Policy = [0.25, 0.25, 0.25, 0.25]
# 状态转移概率: [SlipLeft, MoveFront, SlipRight, SlipBack]
SlipProbs = [0.0, 1.0, 0.0, 0.0]
```

- 按中国人的习惯，定义左、上、右、下顺时针顺序的四个方向。
- 动作空间就由这四个动作组成。当然，在醉汉回家问题中，只有左、右两个动作。
- 初始策略，就是在 4 个方向上随机选择。
- 状态转移概率，就是在动作执行后，是否会出现偏差。举例来说，在冰面向前行走，很有可能冰面太滑而造成向左 0.2、向右 0.1、向前 0.7 的状态转移概率，那么改值就可以写成 [0.2, 0.7, 0.1, 0.0]。注意顺序不能乱。

奖励部分

```
# 每走一步的奖励值，可以是0或者-1
StepReward = 0
# 特殊奖励 from s->s' then get r，其中 s,s' 为状态序号，不是坐标位置
SpecialReward = {
    (0,0):-1,       # s0 -> s0 得到-1奖励
    (2,2):-1,
    ......
    (1,12):+5,
    (3,21):+10
}
```

- StepReward 表示每走一步都可以有 -1 的奖励，或者 0。
- 特殊奖励，比如本例中的碰撞边界得 -1，或者穿越虫洞得 +5 或 +10。这个字典不管智能体是如何从 s_1 到达 s_1 的，也就是忽略了中间的动作选择和状态转移两个步骤，只看起始和终止状态。

特殊移动

```
# 特殊移动，用于处理类似虫洞场景
SpecialMove = {
    (1,LEFT):12,    # 从状态1执行向左的动作会到达状态12
    (1,UP):12,
    (1,RIGHT):12,
    (1,DOWN):12,
    (3,LEFT):20,
    (3,UP):21,
    (3,RIGHT):21,
    (3,DOWN):21
}
# 墙
Blocks = []
```

- 特殊移动用于处理本例中的虫洞场景，比如“(1,LEFT):12”，表示“从状态 1 执行向左的动作会到达状态 12”。
- 墙，用于搭建迷宫类场景。撞墙后一般原地不动。

实现算法

q_π

```
# 根据式 (2.1) 计算 q_pi
def q_pi(p_s_r, gamma, V):
    q = 0
    # 遍历每个转移概率,以计算 q_pi
    for p, s_next, r in p_s_r:
        # math: \sum_{s'} p_{ss'}^a [ r_{ss'}^a + \gamma *  v_{\pi}(s')]
        q += p * (r + gamma * V[s_next])
    return q
```

v_π

```
# 根据式 (5) 计算 v_pi
def v_pi(env: GridWorld, s, gamma, V, Q):
    actions = env.get_actions(s)    # 获得当前状态s下的所有可选动作
    v = 0
    for a, p_s_r in actions:        # 遍历每个动作以计算q值，进而计算v值
        q = q_pi(p_s_r, gamma, V)   # 计算 q_pi 的值
        # math: \sum_a \pi(a|s) q_\pi (s,a)
        v += env.Policy[a] * q      # policy[a]的值为0.25
        # 顺便记录下q(s,a)值,不需要再单独计算一次
        Q[s,a] = q
    return v
```

单数组原地更新迭代算法

```
# 迭代法计算 v_pi
def V_in_place_update(env: GridWorld, gamma, iteration):
    V = np.zeros(env.nS)            # 初始化 V(s)
    Q = np.zeros((env.nS, env.nA))  # 初始化 Q(s,a)
    count = 0   # 计数器，用于衡量性能和避免无限循环
    # 迭代
    while (count < iteration):
        V_old = V.copy()    # 保存上一次的值以便检查收敛性
        # 遍历所有状态 s
        for s in range(env.nS):
            V[s] = v_pi(env, s, gamma, V, Q)    # 计算 v_pi 的值
        # 检查收敛性
        if abs(V-V_old).max() < 1e-4:   # 收敛条件
            break
        count += 1
    # end while
    print(count)
    return V, Q
```

结果

字典

```
状态->动作->转移->奖励 字典:
state = 0
        action = LEFT
         [(1.0, 0, -1)]
        action = UP
         [(1.0, 0, -1)]
        action = RIGHT
         [(1.0, 1, 0)]
        action = DOWN
         [(1.0, 5, 0)]
state = 1
        action = LEFT
         [(1.0, 12, 5)]
        action = UP
         [(1.0, 12, 5)]
```

```
        action = RIGHT
         [(1.0, 12, 5)]
        action = DOWN
         [(1.0, 12, 5)]
......
```

- 在 s_0 状态，向左和向上移动时，都会以 p=1.0 的概率返回，并得到 -1 的奖励。
- 在 s_1 状态，向任意方向移动时，都会以 p=1.0 的概率达到 s_{12} 状态，并得到 +5 的奖励。

v_π

```
迭代次数 =  41
V_pi
[[ 1.66  5.63  4.52  8.73  3.28]
 [ 0.64  2.02  2.3   2.99  1.51]
 [-0.35  0.41  0.7   0.75  0.05]
 [-1.16 -0.56 -0.34 -0.43 -0.97]
 [-1.96 -1.41 -1.22 -1.34 -1.85]]
```

迭代了 41 次收敛，

我们检查一下 v_π 的计算是否正确

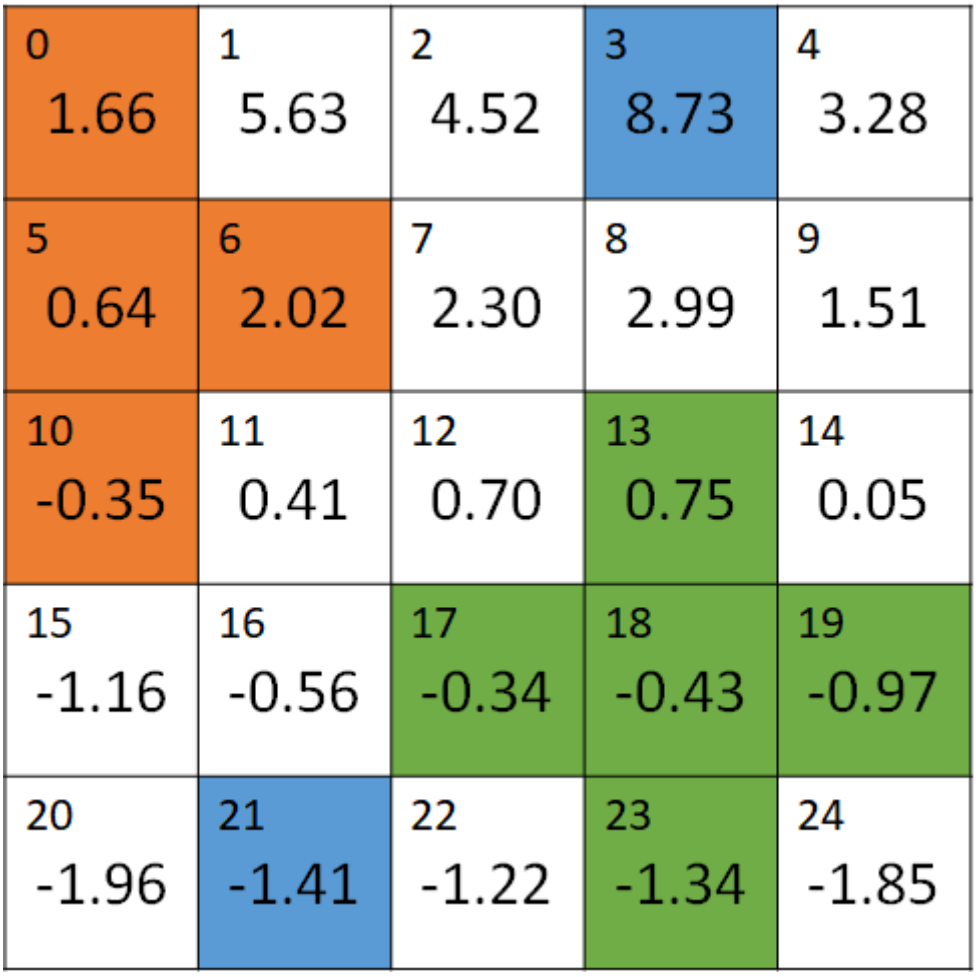

图 6

$v_\pi(s) = \sum_a \pi(a \mid s)\left(\sum_{s'} p^a_{ss'}[r^a_{ss'} + \gamma v_\pi(s')]\right)$

其中，$\pi = 0.25, p = 1.0$，而 r 根据情况有所不同

蓝色 $s=s_3, s'=s_{21}, \pi(a|s_3)=0.25, p^a_{3,21}=1, r^a_{3,21}=10$

$$
\begin{aligned}
v_\pi(s_3) &= \sum_{a\in(L,U,R,D)} \pi(a|s_3)\Big(\sum_{s'=S_{21}} p^a_{3,21}[r^a_{3,21}+0.9v(s_{21})]\Big) \\
&= 4\times 0.25\times\Big(1\times[10+0.9\times(-1.41)]\Big) \\
&\approx 8.73
\end{aligned}
$$

与$v_\pi(s_3)$的值吻合。

橙色和绿色的部分的验证由读者在思考与练习中完成。

再分析一下两个虫洞入口 s_1, s_3 的状态价值函数值

$v_\pi(s_3)=8.73$，小于离开此状态的即时奖励（$R_{t+1}=10$），而 $v_\pi(s_1)=5.63$，大于离开此状态时的即时奖励（$R_{t+1}=5$）。这是为什么呢?

- 因为 s_3 的状态价值函数由其下游状态 s_{21} 决定，而飞船在 s_{21} 有 0.25 的可能出界而得到负的奖励。
- 而 s_1 的下游状态 s_{12} 在中心区域，很不容易出界，状态价值为正数，所以 $v_\pi(s_1)$ 的值要大于即时奖励值。

q_π

```
Q_pi
[[ 0.49  0.49  5.07  0.58]
 [ 5.63  5.63  5.63  5.63]
 [ 5.07  3.06  7.86  2.07]
 [ 8.73  8.73  8.73  8.73]
 [ 7.86  1.95  1.95  1.36]
 [-0.42  1.49  1.82 -0.32]
 [ 0.58  5.07  2.07  0.37]
 [ 1.82  4.06  2.69  0.63]
 [ 2.07  7.86  1.36  0.67]
 [ 2.69  2.95  0.36  0.05]
 [-1.32  0.58  0.37 -1.04]
 [-0.32  1.82  0.63 -0.51]
 [ 0.37  2.07  0.67 -0.31]
 [ 0.63  2.69  0.05 -0.38]
 [ 0.67  1.36 -0.95 -0.87]
 [-2.04 -0.32 -0.51 -1.76]
 [-1.04  0.37 -0.31 -1.27]
 [-0.51  0.63 -0.38 -1.1 ]
 [-0.31  0.67 -0.87 -1.2 ]
 [-0.38  0.05 -1.87 -1.67]
 [-2.76 -1.04 -1.27 -2.76]
 [-1.76 -0.51 -1.1  -2.27]
 [-1.27 -0.31 -1.2  -2.1 ]
 [-1.1  -0.38 -1.67 -2.2 ]
 [-1.2  -0.87 -2.67 -2.67]]
```

q_π 数据解读：

- 一共有 25 行数据，代表了 25 个状态。
- 每行有 4 列数据，代表了每个状态下的四个动作的动作价值函数值，顺序是“左上右下”。

以第一行数据为例：[0.49 0.49 5.07 0.58]，可以看到最大值是 5.07，对应的动作是“右”，于是我们就在图 x 的第 0 个状态中绘制一个向右的箭头，表示智能体应该在此处选择向右走。

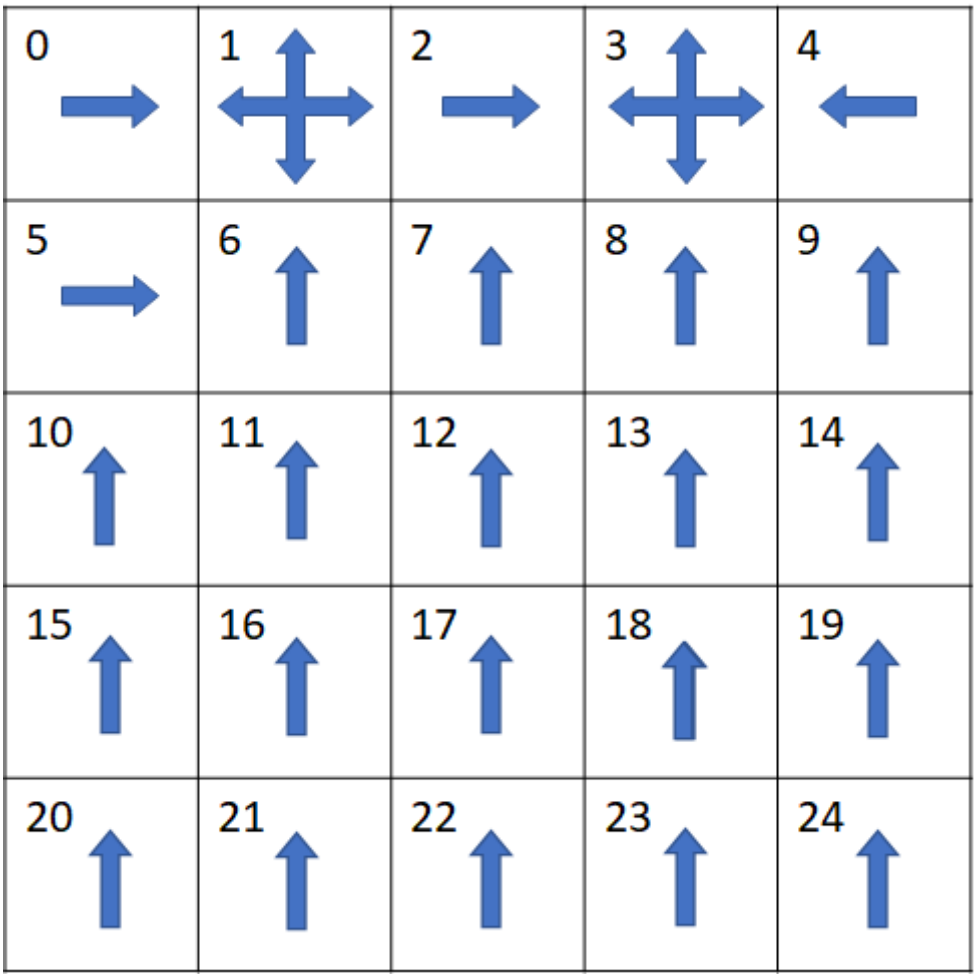

图 6

再看第二行数据：[5.63 5.63 5.63 5.63]，四个值相等，表示可以任意选择一个方向，我们就可以再图 x 的第 1 个状态中绘制一个四个方向的箭头。

针对第二行数据，如果使用 np.argmax([5.63, 5.63, 5.63, 5.63]) 函数，只会返回第一个最大值，这不是我们想要的，应该使用下面的代码来获得所有的最佳动作：

```
best_actions = np.argwhere(self.policy == np.max(self.policy))  #应该返回
[0,1,2,3]
```

绘制完全部 25 个状态的最佳动作后，我们来一起分析一下。

合理的动作：

- 在状态 1 和 3，任意向四个方向移动，都会无条件穿过虫洞，这个没有问题；
- 在状态 2 向右走，因为 v_3 的价值大于 v_1 的价值；
- 在状态 0 向右走到达状态 1，虽然 v_3 更好，但是要绕远（0->5->6->7->2->3），得不偿失；

不合理的动作：

- 在状态 6，是不是可以向右走以便最终到达状态 3 更好呢？
- 在状态 24，如果想最短路径到达状态 3，应该可以选择向上和向左两个方向，在图 x 中只有一个向上的选择。
- 状态 7 和状态 24 情况一样，应该有向上和向右两个可选项，图中只有一个。

分析至此，虽然图 x 中的动作选择大方向没错，但是有些细节值得推敲，也许这个策略组合还不是最佳的。

思考与练习

1. 验证图 x 中 橙色 $v_\pi(s_5)$ 和 绿色 $v_\pi(s_{18})$ 的 v_π 值