动态规划算法

简介

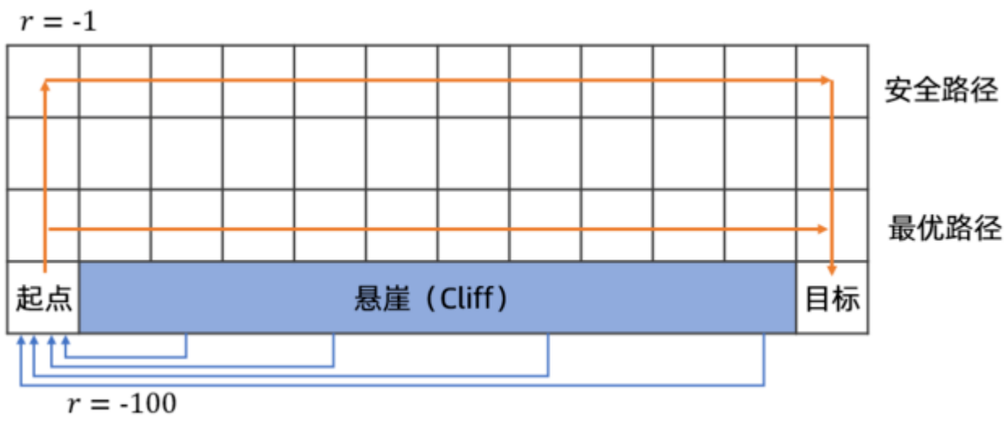
动态规划（Dynamic Programming）是程序设计算法中非常重要的内容，能够高效解决一些经典问题，例如背包问题和最短路径规划。动态规划的基本思想是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。在动态规划中，我们会保存已解决的子问题的答案，而在求解目标问题过程中，如果需要这些子问题答案时，就可以直接利用，避免重复计算。在本节课程内容中，我们就学习如何用动态规划的思想来求解在有限马尔可夫决策过程中的最优策略。

基于动态规划的强化学习算法主要有两种：一是策略迭代（Policy Iteration），二是价值迭代(Value Iteration)。其中，策略迭代有两部分组成：策略评估（Policy Evaluation）和策略提升（Policy Improvement）。具体来说，策略迭代中的策略评估使用贝尔曼期望方程来得到一个策略的状态价值函数，这是一个动态规划的过程。而价值迭代直接使用贝尔曼最优方程来进行动态规划，得到最终的最优状态价值。

不同于我们之前介绍的蒙特卡洛算法和之后将要介绍的时序差分算法，基于动态规划的两大强化学习算法要求我们能事先知道环境的状态转移函数和奖励函数。在这样一个白盒环境中，我们不需要通过智能体和环境的大量交互来学习，可以直接用动态规划求解状态价值。但是，现实中白盒环境很少，这也是动态规划算法的局限所在，我们无法将其运用到很多实际场景中。其次，我们介绍的策略迭代和价值迭代通常只适用于有限马尔可夫决策过程中，即状态空间和动作空间是有限的。

Cliff Walking环境介绍

本节课程中，我们将使用策略迭代和价值迭代来求解Cliff Walking这个环境中的最优策略。接下来我们将简单介绍该环境，示意图如下。



图：Cliff Walking环境示意图

Cliff Walking是一个非常经典的强化学习环境，它要求一个智能体从起点出发，避开悬崖行走，最终到达目标位置。如图所示，有一个$4 * 12$的网格世界，每一个网格是一个状态，起点是左下角的状态，目标是右下角的状态。智能体在每一个状态都可以采取4种动作：上,下,左,右，如果采取动作后触碰到边界墙壁则状态不发生改变，否则就会相应到达下一个状态。其中有一段悬崖，智能体到达目标状态或掉入悬崖都会结束并回到起点，也就是说它们是终止状态。每走一步的奖励是-1，掉入悬崖的奖励是-100。

接下来一起来看一看Cliff Walk环境的代码吧。

In [1]:

```
import copy

class CliffWalkingEnv:
    """ Cliff Walking环境"""
    def __init__(self, ncol=12, nrow=4):
        self.ncol = ncol # 定义环境的宽
        self.nrow = nrow # 定义环境的高
        self.P = self.createP() # 转移矩阵P[state][action] = [(p, next_state, reward, done)]，包含下一个状态和奖励

    def createP(self):
        P = [[[] for j in range(4)] for i in range(self.nrow * self.ncol)] # 初始化
        change = [[0, -1], [0, 1], [-1, 0], [1, 0]] # 4 种动作, 0:上, 1:下, 2:左, 3:右。原点(0,0)定义在左上角
        for i in range(self.nrow):
            for j in range(self.ncol):
                for a in range(4):
                    if i == self.nrow - 1 and j > 0:  # 位置在悬崖或者终点，因为无法继续交互，任何动作奖励都为0
                        P[i * self.ncol + j][a] = [(1, i * self.ncol + j, 0, True)]
                        continue
                    # 其他位置
                    next_x = min(self.ncol - 1, max(0, j + change[a][0]))
                    next_y = min(self.nrow - 1, max(0, i + change[a][1]))
                    next_state = next_y * self.ncol + next_x
                    reward = -1
                    done = False
                    if next_y == self.nrow - 1 and next_x > 0: # 下一个位置在悬崖或者终点
                        done = True
                        if next_x != self.ncol - 1: # 下一个位置在悬崖
                            reward = -100
                    P[i * self.ncol + j][a] = [(1, next_state, reward, done)]
        return P
```

策略迭代

策略迭代是策略评估和策略提升这两个过程的不断循环交替，直至最后得到最优策略。我们接下来分别进行详细介绍。

策略评估

策略评估这一过程用来计算一个策略的状态价值函数。回顾一下之前学习的贝尔曼期望方程：

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \left(r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_\pi(s') \right)$$

其中$\pi(a|s)$是策略π在状态s下采取动作a的概率。我们看到，当我们知道奖励函数和状态转移函数时，我们可以根据下一个状态的价值来计算当前状态的价值。于是，根据动态规划的思想，我们可以将计算下一个可能状态的价值当成一个子问题，计算当前状态的价值看成当前问题。在得知子问题的解后，我们就可以求解当前问题。更一般的，我们考虑所有的状态，就变成了我们用上一轮的状态价值函数来计算当前这一轮的状态价值函数。即

$$V_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left(r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_k(s') \right)$$

我们可以选定任意初始值V_0。根据贝尔曼方程，我们得知$V_k = V_\pi$是以上更新公式的一个不动点（fixed point）。事实上，我们可以证明当$k \to \infty$时，序列$\{V_k\}$会收敛到V_π。所以我们可以据此来计算得到一个策略的状态价值函数。在实际实现过程中，在某一轮如果$\max_{s \in \mathcal{S}} |V_{k+1}(s) - V_k(s)|$的值非常小时，我们会提前结束策略评估。这样做可以提升效率，并且得到的价值函数也非常接近真实的价值函数。

策略提升

在用策略评估计算得到当前策略的状态价值函数之后，我们可以据此来改进该策略，这个步骤称为策略提升。假设此时对于策略π，我们已经知道其策略函数V_π，也就是我们知道了从每一个状态s出发一直根据策略π最终得到的期望回报。但是我们要如何改变策略来获得在状态s下更高的期望回报呢？假设我们在状态s下采取动作a然后之后的动作依旧遵循策略π，此时得到

的期望回报其实就是动作价值$Q_\pi(s,a)$。如果我们有$Q_\pi(s,a)>V_\pi(s)$，则说明在状态s下采取动作a会比原来的策略$\pi(a|s)$得到更高的期望回报。以上只是针对一个状态。现在假设存在一个确定性策略π'，在任意一个状态s下，都满足

$$Q_\pi(s,\pi'(s))\geq V_\pi(s),$$

于是在任意状态s下，我们有

$$V_{\pi'}(s)\geq V_\pi(s).$$

这便是策略提升定理（Policy Improvement Theorem）。于是我们可以直接贪心地在每一个状态选择动作价值最大的动作，也就是

$$\pi'(s)=\arg\max_a Q_\pi(s,a)=\arg\max_a\{r(s,a)+\gamma\sum_{s'}p(s'|s,a)V_\pi(s')\}$$

我们发现构造的贪心策略π'满足策略提升定理的条件，所以策略π'能够比策略π更好或者与其一样好。这个根据贪心法选取动作得到新的策略的过程称为策略提升。当策略提升之后得到的策略π'和之前的策略π一样时，此时π和π'就是最优策略。

策略提升定理的证明：

$$\begin{aligned}
V_\pi(s)&\leq Q_\pi(s,\pi'(s))\\
&=\mathbb{E}_{\pi'}[R_t+\gamma V_\pi(S_{t+1})|S_t=s]\\
&\leq\mathbb{E}_{\pi'}[R_t+\gamma Q_\pi(S_{t+1},\pi'(S_{t+1}))|S_t=s]\\
&=\mathbb{E}_{\pi'}[R_t+\gamma R_{t+1}+\gamma^2V_\pi(S_{t+2})|S_t=s]\\
&\leq\mathbb{E}_{\pi'}[R_t+\gamma R_{t+1}+\gamma^2R_{t+2}+\gamma^3V_\pi(S_{t+3})|S_t=s]\\
&\vdots\\
&\leq\mathbb{E}_{\pi'}[R_t+\gamma R_{t+1}+\gamma^2R_{t+2}+\gamma^3R_{t+3}+\cdots|S_t=s]\\
&=V_{\pi'}(s)
\end{aligned}$$

策略迭代算法

总体来说，策略迭代算法为我们对当前的策略进性策略评估，得到其状态价值函数，然后用该状态价值函数根据策略提升来得到一个更好的新策略，然后再继续评估新策略再提升策略，以此往复，直至最后收敛到最优策略：

$$\pi_0\xrightarrow{\text{策略评估}}V_{\pi_0}\xrightarrow{\text{策略提升}}\pi_1\xrightarrow{\text{策略评估}}V_{\pi_1}\xrightarrow{\text{策略提升}}\pi_2\xrightarrow{\text{策略评估}}\dots\xrightarrow{\text{策略提升}}\pi_*$$

结合策略评估和策略提升，我们得到以下策略迭代算法。

- 随机初始化策略$\pi(s)$和价值函数$V(s)$
- 进行如下**策略评估**循环:
  - $\Delta\leftarrow 0$
  - 对于每一个状态 $s\in\mathcal{S}$:
    - $v\leftarrow V(S)$
    - $V(s)\leftarrow r(s,\pi(s))+\gamma\sum_{s'}p(s'|s,\pi(s))V(s')$
    - $\Delta\leftarrow\max(\Delta,|v-V(s)|)$
  - 直到 $\Delta<\theta$，结束循环
- $\pi_{\text{old}}\leftarrow\pi$
- 对于每一个状态 $s\in\mathcal{S}$:
  - $a\leftarrow\pi(s)$
  - $\pi(s)\leftarrow\arg\max_a r(s,a)+\gamma\sum_{s'}p(s'|s,a)V(s')$
- 若$\pi_{\text{old}}=\pi$, 则停止算法并返回 V 和 π; 否则转到策略评估循环

我们现在来看一下策略迭代的代码。

In [2]:

```
class PolicyIteration:
    """ 策略迭代 """
    def __init__(self, env, theta, gamma):
        self.env = env
        self.v = [0] * self.env.ncol * self.env.nrow  # 初始化价值为0
        self.pi = [[0.25, 0.25, 0.25, 0.25] for i in range(self.env.ncol * self.env.nrow)]  # 初始化为均匀随机策略
        self.theta = theta  # 策略评估收敛阈值
        self.gamma = gamma  # 折扣因子

    def policy_evaluation(self): # 策略评估
        cnt = 1 # 计数器
        while 1:
            max_diff = 0
            new_v = [0] * self.env.ncol * self.env.nrow
            for s in range(self.env.ncol * self.env.nrow):
                qsa_list = []
                for a in range(4):
                    qsa = 0
                    for res in self.env.P[s][a]:
                        p, next_state, r, done = res
                        qsa += p * (r + self.gamma * self.v[next_state] * (1-done))  # 本章节环境比较特殊，奖励和下一个状态有关，所以需要和状态转移
                    qsa_list.append(self.pi[s][a] * qsa)
                new_v[s] = sum(qsa_list)  # 状态价值函数和动作价值函数之间的关系
                max_diff = max(max_diff, abs(new_v[s] - self.v[s]))
            self.v = new_v
            if max_diff < self.theta: break
            cnt += 1
        print("策略评估进行%d轮后完成" % cnt)

    def policy_improvement(self): # 策略提升
        for s in range(self.env.nrow * self.env.ncol):
            qsa_list = []
            for a in range(4):
                qsa = 0
                for res in self.env.P[s][a]:
                    p, next_state, r, done = res
                    qsa += p * (r + self.gamma * self.v[next_state] * (1-done))
                qsa_list.append(qsa)
            maxq = max(qsa_list)
            cntq = qsa_list.count(maxq)
            self.pi[s] = [1/cntq if q == maxq else 0 for q in qsa_list]  # 让相同的动作价值均分概率
        print("策略提升完成")
        return self.pi

    def policy_iteration(self): # 策略迭代
        while 1:
            self.policy_evaluation()
            old_pi = copy.deepcopy(self.pi) # 将列表进行深拷贝，方便接下来进行比较
            new_pi = self.policy_improvement()
            if old_pi == new_pi: break
```

现在我们已经写好了环境代码和策略迭代代码。为了更好的展现最终的策略，我们下面增加一个打印策略的函数，来打印当前策略每个状态下的价值以及会采取的动作。对于打印出来的动作，我们用"^o<o"表示等概率采取向左和向上两种动作，"ooo>"表示在当前状态只采取向右动作。

In [3]:

```
def print_agent(agent, action_meaning, disaster=[], end=[]):
    print("状态价值：")
    for i in range(agent.env.nrow):
        for j in range(agent.env.ncol):
            print('%6.6s' % ('%.3f' % agent.v[i * agent.env.ncol + j]), end=' ') # 为了输出美观，保持输出6个字符
        print()

    print("策略：")
    for i in range(agent.env.nrow):
        for j in range(agent.env.ncol):
            if (i * agent.env.ncol + j) in disaster: # 一些特殊的状态，例如Cliff Walking中的悬崖
                print('****', end=' ')
            elif (i * agent.env.ncol + j) in end: # 终点
                print('EEEE', end=' ')
            else:
                a = agent.pi[i * agent.env.ncol + j]
                pi_str = ''
                for k in range(len(action_meaning)):
                    pi_str += action_meaning[k] if a[k] > 0 else 'o'
                print(pi_str, end=' ')
        print()
```

In [4]:

```
env = CliffWalkingEnv()
action_meaning = ['^', 'v', '<', '>']
theta = 0.001
gamma = 0.9
agent = PolicyIteration(env, theta, gamma)
agent.policy_iteration()
print_agent(agent, action_meaning, list(range(37, 47)), [47])
```

```
策略评估进行60轮后完成
策略提升完成
策略评估进行72轮后完成
策略提升完成
策略评估进行44轮后完成
策略提升完成
策略评估进行12轮后完成
策略提升完成
策略评估进行1轮后完成
策略提升完成
状态价值：
-7.712 -7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710
-7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900
-7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900 -1.000
-7.458  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
策略：
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ovoo
^ooo **** **** **** **** **** **** **** **** **** **** EEEE
```

我们发现经过五次策略评估和策略提升的循环迭代，策略收敛了。此时获得的策略也已经打印出来了，我们可以用贝尔曼最优方程去检验每一个状态的价值，我们可以发现最终输出的策略的确是最有策略。

价值迭代

策略迭代中的策略评估需要很多轮才能收敛得到某一策略的状态函数，这需要很大的计算量。我们是否必须要完全等到策略评估完成？试想一下，可能出现这样的情况：虽然状态价值函数还没收敛，但是不论接下来怎么更新状态价值，策略提升得到的可能是同一个策略。更特殊一点，如果我们只在策略评估中进行一轮价值的更新，然后直接根据更新后的价值进行策略提升，这样是否可以呢？答案是肯定的，这其实就是我们将要学习的价值迭代算法，我们可以认为它是一种策略评估只进行了一次更新的策略迭代。需要注意的是，价值迭代中不存在一个显式的策略，我们只维护一个状态价值函数。

确切来说，价值迭代可以看成一种动态规划过程，它利用的是贝尔曼最优方程：

$$V_*(s) = \max_{a \in \mathcal{A}} \{r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_*(s')\}$$

将其写成迭代更新的方式为

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \{r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s')\}$$

价值迭代便是按照以上更新方式进行。等到V_{k+1}和V_k一样时，它就是贝尔曼最优方程的不动点，此时对应着最优状态价值函数V_*。然后我们从中恢复出最优策略即可：$\pi(s) = \arg\max_a \{r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_{k+1}(s')\}$。

价值迭代算法

下面我们更加详细的介绍价值迭代的算法流程。

- 随机初始化$V(s)$
- 进行如下**价值迭代**循环:
  - $\Delta \leftarrow 0$
  - 对于每一个状态 $s \in \mathcal{S}$:
    - $v \leftarrow V(s)$
    - $V(s) \leftarrow \max_a r(s,a) + \gamma \sum_{s'} p(s'|s,a) V(s')$
    - $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  - 若$\Delta < \theta$，结束循环
- 返回一个确定性策略 $\pi(s) = \arg\max_a \{r(s,a) + \gamma \sum_{s'} p(s'|s,a) V(s')\}$

我们现在来实现价值迭代的代码。

In [5]:

```
class ValueIteration:
    """ 价值迭代 """
    def __init__(self, env, theta, gamma):
        self.env = env
        self.v = [0] * self.env.ncol * self.env.nrow  # 初始化价值为0
        self.theta = theta  # 价值收敛阈值
        self.gamma = gamma
        self.pi = [None for i in range(self.env.ncol * self.env.nrow)] # 价值迭代结束后得到的策略

    def value_iteration(self):
        cnt = 0
        while 1:
            max_diff = 0
            new_v = [0] * self.env.ncol * self.env.nrow
            for s in range(self.env.ncol * self.env.nrow):
                qsa_list = []
                for a in range(4):
                    qsa = 0
                    for res in self.env.P[s][a]:
                        p, next_state, r, done = res
                        qsa += p * (r + self.gamma * self.v[next_state] * (1-done))
                    qsa_list.append(qsa) # 这一行和下一行是和策略迭代的主要区别
                new_v[s] = max(qsa_list)
                max_diff = max(max_diff, abs(new_v[s] - self.v[s]))
            self.v = new_v
            if max_diff < self.theta: break
            cnt += 1
        print("价值迭代一共进行%d轮" % cnt)
        self.get_policy()

    def get_policy(self): # 根据价值函数导出一个贪心策略
        for s in range(self.env.nrow * self.env.ncol):
            qsa_list = []
            for a in range(4):
                qsa = 0
                for res in self.env.P[s][a]:
                    p, next_state, r, done = res
                    qsa += r + p * self.gamma * self.v[next_state] * (1-done)
                qsa_list.append(qsa)
            maxq = max(qsa_list)
            cntq = qsa_list.count(maxq)
            self.pi[s] = [1/cntq if q == maxq else 0 for q in qsa_list]  # 让相同的动作价值均分概率
```

In [6]:

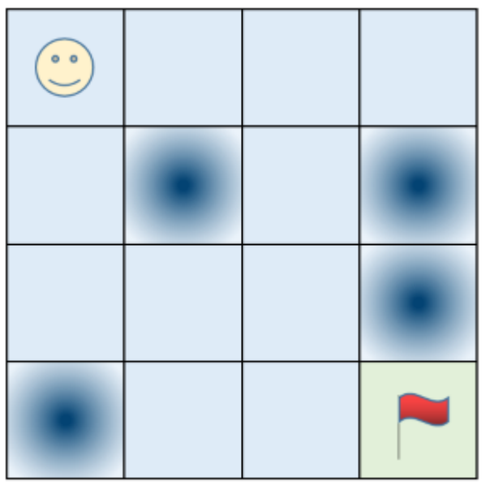
```
env = CliffWalkingEnv()
action_meaning = ['^', 'v', '<', '>']
theta = 0.001
gamma = 0.9
agent = ValueIteration(env, theta, gamma)
agent.value_iteration()
print_agent(agent, action_meaning, list(range(37, 47)), [47])
```

```
价值迭代一共进行14轮
状态价值：
-7.712 -7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710
-7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900
-7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900 -1.000
-7.458  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
策略：
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ovoo
^ooo **** **** **** **** **** **** **** **** **** **** EEEE
```

价值迭代总共用了14轮而策略迭代中策略评估总共用了263轮，可以发现价值迭代中循环次数远少于策略迭代。

Frozen Lake环境

除了Cliff Walking环境，我们还提供了另外一个环境为Frozen Lake。该环境的状态空间和动作空间是有限的，我们在Frozen Lake环境也尝试策略迭代和价值迭代算法，帮助大家更好地理解这两个算法。



Frozen Lake是OpenAI Gym库中的一个环境。OpenAI Gym库中包含了很多有名环境，例如Atari和MuJoCo，并且它支持我们定制自己的环境。在之后的章节中，我们还会使用到更多来自OpenAI Gym库中的环境。具体来说，Frozen Lake环境和Cliff Walking一样也是一个方格世界，大小为$4 * 4$。每一个方格是一个状态，起点状态(S)在左上角，终点状态(G)在右下角，中间还有若干冰洞(H)。在每一个状态都可以采取上下左右4个动作。由于是冰面，每次行走有一定的概率滑行到附近的其它状态，并且到达冰洞和终点会提前结束。每一步奖励是0，到达终点奖励是1。

我们先用代码调用一下gym中的FrozenLake-v0环境，并且简单查看一下环境信息，然后找出冰洞和终点状态。

In [7]:

```
import gym
env = gym.make("FrozenLake-v0")  # 创建环境
env = env.unwrapped  # 解封装才能访问状态转移矩阵P
env.render()  # 环境渲染，通常是弹窗显示或打印出可视化的环境

holes = set()
ends = set()
for s in env.P:
    for a in env.P[s]:
        for s_ in env.P[s][a]:
            if s_[2] == 1.0: # 获得奖励为1，代表是终点
                ends.add(s_[1])
            if s_[3] == True:
                holes.add(s_[1])
holes = holes - ends
print("冰洞的索引:", holes)
print("终点的索引", ends)

for a in env.P[14]:  # 查看终点左边一格的状态转移信息
    print(env.P[14][a])
```

```
SFFF
FHFH
FFFH
HFFG
冰洞的索引: {11, 12, 5, 7}
终点的索引 {15}
[(0.3333333333333333, 10, 0.0, False), (0.3333333333333333, 13, 0.0, False), (0.3333333333333333, 14, 0.0, False)]
[(0.3333333333333333, 13, 0.0, False), (0.3333333333333333, 14, 0.0, False), (0.3333333333333333, 15, 1.0, True)]
[(0.3333333333333333, 14, 0.0, False), (0.3333333333333333, 15, 1.0, True), (0.3333333333333333, 10, 0.0, False)]
[(0.3333333333333333, 15, 1.0, True), (0.3333333333333333, 10, 0.0, False), (0.3333333333333333, 13, 0.0, False)]
```

首先我们发现冰洞的索引是$\{5, 7, 11, 12\}$，这说明在Frozen Lake这个环境中，原点（第一个状态）的定义是在左上角，这和Cliff Walking环境一样。其次，根据第15个状态（也即终点左边一格，数组下标索引为14）的信息，我们可以看到每个动作都会等概率“滑行”到3种可能的结果，这和Cliff Walking环境是不一样的。我们接下来先用策略迭代来试一下。

In [8]:

```
action_meaning = ['<', 'v', '>', '^'] # 这个动作意义是gym库中对FrozenLake这个环境事先规定好的
theta = 1e-5
gamma = 0.9
agent = PolicyIteration(env, theta, gamma)
agent.policy_iteration()
print_agent(agent, action_meaning, [5, 7, 11, 12], [15])
```

```
策略评估进行25轮后完成
策略提升完成
策略评估进行58轮后完成
策略提升完成
状态价值:
 0.069  0.061  0.074  0.056
 0.092  0.000  0.112  0.000
 0.145  0.247  0.300  0.000
 0.000  0.380  0.639  0.000
策略:
<ooo ooo^ <ooo ooo^
<ooo **** <o>o ****
ooo^ ovoo <ooo ****
**** oo>o ovoo EEEE
```

这个最优策略很看上去比较反直觉，但其实原因是因为这是一个会随机滑向其他状态的冰冻湖面。例如在终点左侧的状态，如果智能体采取向右的动作，它有可能会滑到冰洞，所以此时采取向下的动作是更为保险的，并且有一定概率能够滑到终点。然后我们来试一下价值迭代。

In [9]:

```
action_meaning = ['<', 'v', '>', '^']
theta = 1e-5
gamma = 0.9
agent = ValueIteration(env, theta, gamma)
agent.value_iteration()
print_agent(agent, action_meaning, [5, 7, 11, 12], [15])
```

```
价值迭代一共进行60轮
状态价值:
 0.069  0.061  0.074  0.056
 0.092  0.000  0.112  0.000
 0.145  0.247  0.300  0.000
 0.000  0.380  0.639  0.000
策略:
<ooo ooo^ <ooo ooo^
<ooo **** <o>o ****
ooo^ ovoo <ooo ****
**** oo>o ovoo EEEE
```

我们发现价值迭代的结果和策略迭代的结果完全一致，这也互相验证了各自的结果。

总结

我们在本章内容中学习了强化学习中两个经典的动态规划算法：策略迭代和价值迭代，它们都能用来求解最优价值函数和最优策略。动态规划的主要思想是对所有状态利用贝尔曼方程进行更新。需要注意的是，在利用贝尔曼方程进行更新时，我们会用到马尔可夫决策过程中的奖励函数和状态转移函数。那如果智能体无法事先得知奖励函数和状态转移函数，只能通过和环境进行交互采样到（状态-动作-奖励-下一状态）这样的数据，我们将在之后的章节中介绍如何求解这种情况下的最优策略。

扩展阅读：收敛性证明

策略迭代

策略迭代的过程如下：

$$\pi_0 \xrightarrow{\text{策略评估}} V_{\pi_0} \xrightarrow{\text{策略提升}} \pi_1 \xrightarrow{\text{策略评估}} V_{\pi_1} \xrightarrow{\text{策略提升}} \pi_2 \xrightarrow{\text{策略评估}} \cdots \xrightarrow{\text{策略提升}} \pi_*$$

并且根据策略提升定理，我们知道更新后的策略的价值函数满足单调性，即$V_{\pi_{k+1}} \geq V_{\pi_k}$。所以只要所有可能的策略个数是有限的，那么策略迭代就能收敛到最优策略。假设MDP的状态空间大小为$|\mathcal{S}|$，动作空间大小为$|\mathcal{A}|$，此时所有可能的策略个数为$|\mathcal{A}|^{|\mathcal{S}|}$，是有限个。所以策略迭代能够在有限步能找到其中的最优策略。

价值迭代

价值迭代的更新公式为

$$V_{k+1}(s)=\max_{a\in\mathcal{A}}\{r(s,a)+\gamma\sum_{s'\in\mathcal{S}}p(s'|s,a)V_k(s')\}$$

我们将其定义为一个**贝尔曼最优算子$\mathcal{T}$**:

$$V_{k+1}(s)=\mathcal{T}V_k(s)=\max_{a\in\mathcal{A}}\{r(s,a)+\gamma\sum_{s'\in\mathcal{S}}p(s'|s,a)V_k(s')\}$$

然后我们引入**压缩算子**（Contraction Operator）的定义：若O是一个算子，如果满足$||OV-OV'||_q\leq||V-V'||_q$条件，则我们称$O$是一个压缩算子。其中$||x||_q$表示$x$的$L_q$范数，包括我们将会用到的无穷范数$||x||_\infty=\max_i|x_i|$

我们接下来证明当$\gamma<1$时，贝尔曼最优算子$\mathcal{T}$是一个γ-压缩算子。

$$\begin{aligned}
||\mathcal{T}V-\mathcal{T}V'||_\infty&=\max_{s\in\mathcal{S}}|\max_{a\in\mathcal{A}}\{r(s,a)+\gamma\sum_{s'\in\mathcal{S}}p(s'|s,a)V(s')\}-\max_{a'\in\mathcal{A}}\{r(s,a')+\gamma\sum_{s'\in\mathcal{S}}p(s'|s,a')V'(s')\}|\\
&\leq\max_{s,a}|r(s,a)+\gamma\sum_{s'\in\mathcal{S}}p(s'|s,a)V(s')-r(s,a)-\gamma\sum_{s'\in\mathcal{S}}p(s'|s,a)V'(s')|\\
&=\gamma\max_{s,a}|\sum_{s'\in\mathcal{S}}p(s'|s,a)(V(s')-V'(s'))|\\
&\leq\gamma\max_{s,a}|\sum_{s'\in\mathcal{S}}p(s'|s,a)|\max_{s'}|V(s')-V'(s')|\\
&=\gamma||V-V'||_\infty
\end{aligned}$$

于是我们有

$$||V_{k+1}-V_*||_\infty=||\mathcal{T}V_k-\mathcal{T}V_*||_\infty\leq\gamma||V_k-V_*||_\infty\leq\cdots\leq\gamma^{k+1}||V_0-V_*||_\infty$$

这意味着，在$\gamma\leq1$的情况下，当k越来越大，即迭代次数越来越多的时候，V_k会越来越接近V_*，即$\lim_{k\to\infty}V_k=V_*$。至此，价值迭代的收敛性得到证明。