该算法已经整理到博客
https://blog.csdn.net/weixin_42763696/article/details/105539737中

Question 6 Q Learning

由于现实世界中并不能获取全部的state以及全部的action，因此值迭代方法在很多问题上还是会有局限性。这时用到的就是Q Learning方法了，对于上述两个问题他会这样解决：

1. 计算的时候不会遍历所有的格子，只管当前状态，当前格子的reward值
2. 不会计算所有action的reward，每次行动时，只选取一个action，只计算这一个action的reward

这样的规则也说明了需要大量的尝试，才能学习出比较好的结果。Q Learning的公式如下：

$$q_\pi(s,a) = q_\pi(s,a) + \alpha[R + \gamma \underset{a'}{max}\, q_\pi(s',a') - q_\pi(s,a)]$$

整理后得到

$$q_\pi(s,a) = (1-\alpha)q_\pi(s,a) + \alpha[R + \gamma \underset{a'}{max}\, q_\pi(s',a')]$$

从左到右拆解开来分析

- $q_\pi(s,a)$表示的是在s时执行a的reward值之和，包括了经验reward值和新的reward值的相加。
- $(1-\alpha)q_\pi(s,a)$表示的是经验reward，即`学习率*之前学习到的执行该action的reward`。可以看到学习速率α越大，保留之前训练的效果就越少。
- $\alpha[R + \gamma \underset{a'}{max}\, q_\pi(s',a')]$就是新的reward值了，下面逐步拆解。
- $\gamma \underset{a'}{max}\, q_\pi(s',a')$是计算下一个state'中最大的reward值，这个称之为“记忆奖励”。因为在之前某次到达state'的时候，保存了四个方向(a')的reward值，通过“回忆”，想起来自己之前在state'上能收获的最大好处，就可以直接影响在当前state时reward的计算。γ是用来增加or减少state'的影响的，γ越大，智能体就会越重视以往经验，越小，就只重视眼前利益（R）。
- R是执行了action后的reward，比如在终点处执行exit，获得+1/-1的reward。

编写代码的时候，需要在`update`函数中体现上述思想。接下来分别实现函数

`getQValue(state,action)`函数

返回Q Value的值，直接return就可以，代码如下

```
def getQValue(self, state, action):
    return self.Q[(state, action)]
```

`computeValueFromQValues(state)`函数

该函数是通过QValue返回最大的reward，因此需要遍历四个reward，最终得到最大值

```
def computeValueFromQValues(self, state):
    actions = self.getLegalActions(state)
    if len(actions) == 0:
        return 0
    # 保存成列表
    values = [self.getQValue(state, action) for action in
actions]
    return max(values)
```

`computeActionFromQValues(state)`函数

和上一个函数一样，只不过这里返回的是最大Action

```
def computeActionFromQValues(self, state):

    actions = self.getLegalActions(state)
    if len(actions) == 0:
        return 0
    max_action = float('-inf')
    best_action = actions[0]
    # 记录最大action
    for action in actions:
        if max_action < self.getQValue(state, action):
            max_action = self.getQValue(state, action)
            best_action = action
    return best_action
```

`getAction(state)`函数

此时要返回的action应该是最大的action

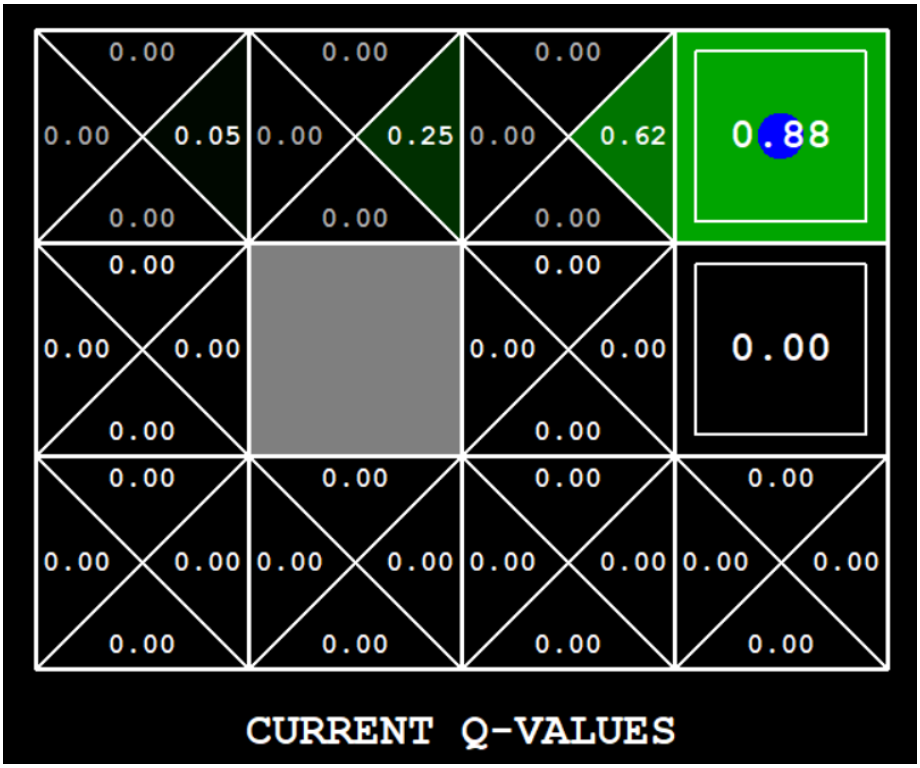
```
def getAction(self, state):
    legalActions = self.getLegalActions(state)
     action = None
     if len(legalActions) == 0:
         return action
    return self.computeActionFromQValues(state)
```

update(state, action, nextState, reward)`函数

这里就是要通过公式计算，更新Q Value值

```
def update(self, state, action, nextState, reward):
    sample = reward + self.discount */
            self.getValue(nextState)
    mid = self.Q[(state,action)]
    # 公式
    self.Q[(state,action)] = (1 - self.alpha) * /
        self.getQValue(state,action) + self.alpha * sample
```

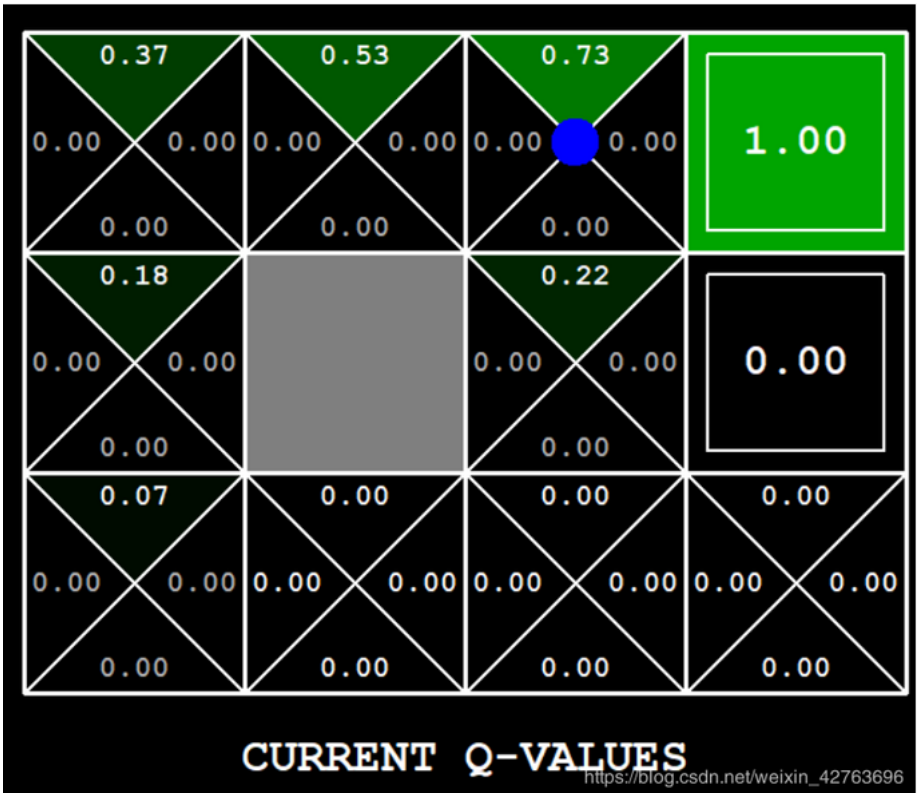
搭建完毕后，就可以计算每一个action的reward了：



按照相同路径走4次后的学习结果

Question 7 Epsilon Greedy

上述的算法看上去可以在每次动作都选择到最佳的动作，但在使用上述算法让智能体去学习Grid World的时候，会遇到下图的问题



最佳action的选择不对劲

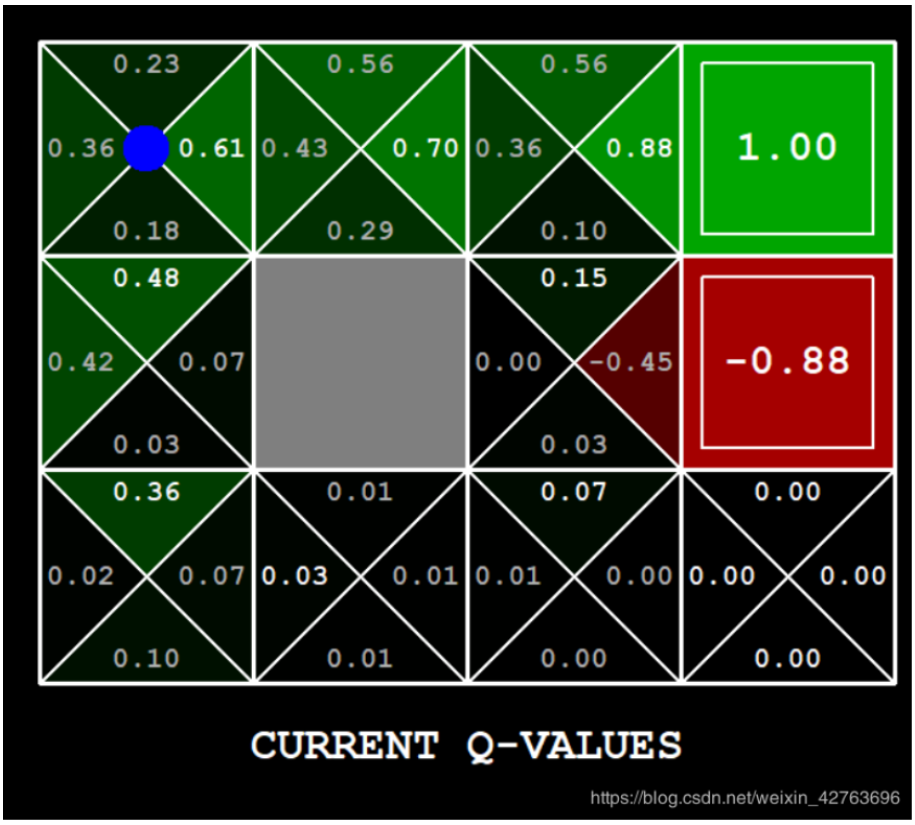
上述是迭代了好多次后智能体学会的最佳action选择，可以看到这个结果还是很有问题的，毕竟在第一行的时候，很明显朝右的方向是最佳选择，但学习出的都是朝上的方向，这里问题出在action的选择上。

在Q Learning的方法中，我们选择每次reward最大的方向，这样就会陷入局部最优，即当向上的action的reward>0时，我就不会再去看别的方向的reward，即使向右走的reward要更大

这里的优化方法就是使用Epsilon Greedy，即以Epsilon的概率去选取reward最大的action，或者随机的action，代码如下：

```
def getAction(self, state):
    legalActions = self.getLegalActions(state)
    action = None
    if len(legalActions) == 0:
        return action
    # 当小于epsilon的时候就用最大值
    if util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    # 否则就用随机action
    return self.computeActionFromQValues(state)
```

这样跑出来的结果就正常多了：



完结，撒花~

111171 董安宁