Project 1 价值迭代(Value Iterator)

方程解析

价值迭代的方程为$V_{i+1}(s) = \underset{a}{max} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma Vi(s')]$，可以拆解开来看：

- $T(s,a,s')$表示从状态s，执行动作a后，到s'的概率。
- $R(s,a,s') + \gamma Vi(s')$表示到达s'后能获得的奖励值。$R(s,a,s')$表示的是每走一步的奖励值；$\gamma Vi(s')$就是用GAMMA乘在s'处的奖励值，使得奖励递减。
- $\sum_{s'} T(s,a,s')[R(s,a,s') + \gamma Vi(s')]$，可以看到是对s'进行求和，即对执行每个action时所能达到的所有s'的奖励进行求和。
- $\underset{a}{max} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma Vi(s')]$，可以看到是对求a的max，也就是哪个action能得到最大的奖励。最终选出最合适的Action。

值迭代函数

手撕公式以后就好写代码了，首先是`runValueIteration`的函数，是用来进行迭代的主函数，用一个for循环来控制迭代次数，每次迭代都执行相同的步骤，即遍历所有的grid，计算并更新每个格子的Value，最终记录下来，用于下一次迭代的计算。代码如下所示：

```
def runValueIteration(self):
    # 用for循环控制迭代次数
    for i in range(self.iterations):
        for state in self.mdp.getStates():
            if self.mdp.isTerminal(state) == False:
                action = self.getAction(state)
                self.values[state] =
self.computeQValueFromValues(state,action)
```

这里用到了两个在下面定义的函数，分别是计算最佳Action的`getAction(state)`和计算最大Reward的`computeQValueFromValues(state,action)`，接下来分别介绍。

`computeQValueFromValues(state,action)`函数

该函数的功能是用来计算Q Value的，输入的参数是state(哪块格子)以及action(要执行的动作)，执行如下操作：

- 记录当前action导致的可能状态变换
- 将所有可能状态变换的reward进行求和
- 返回求和结果

因此代码如下：

```
def computeQValueFromValues(self, state, action):
    q = 0
    newStates =
self.mdp.getTransitionStatesAndProbs(state, action)
    for x in newStates:
        q += x[1] * (self.mdp.getReward(state, action,
x[0]) + self.discount * self.values[x[0]])
```

目的是计算出在当前state执行特定action时，能得到的期望奖励值。

`getAction(state)`函数

该函数的目的是为了得到在当前state下，应该执行的最好的action，为此需要执行以下步骤：

- 遍历当前state的所有action
- 计算每个action的Q Value
- 判断哪个action的Q Value最大，并返回该值

代码如下：

```
def computeActionFromValues(self, state):
    #print('state',state)
    newDict = util.Counter()
    for action in self.mdp.getPossibleActions(state):
        newDict[action] =
self.computeQValueFromValues(state, action)
    return newDict.argMax()
```

当上述代码填充完毕后，便可得到迭代的结果了，如下图所示：

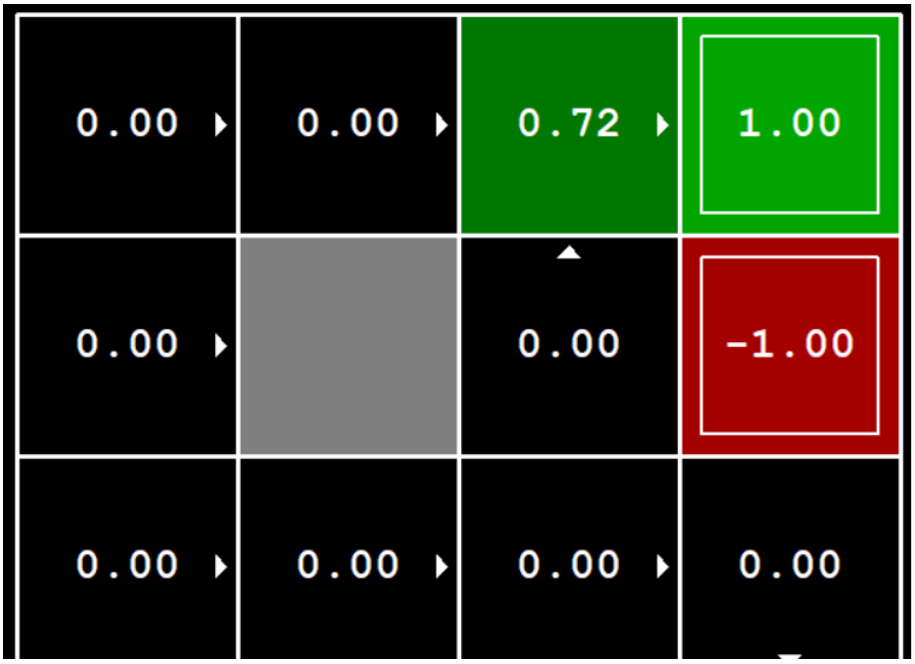

图1 迭代2次

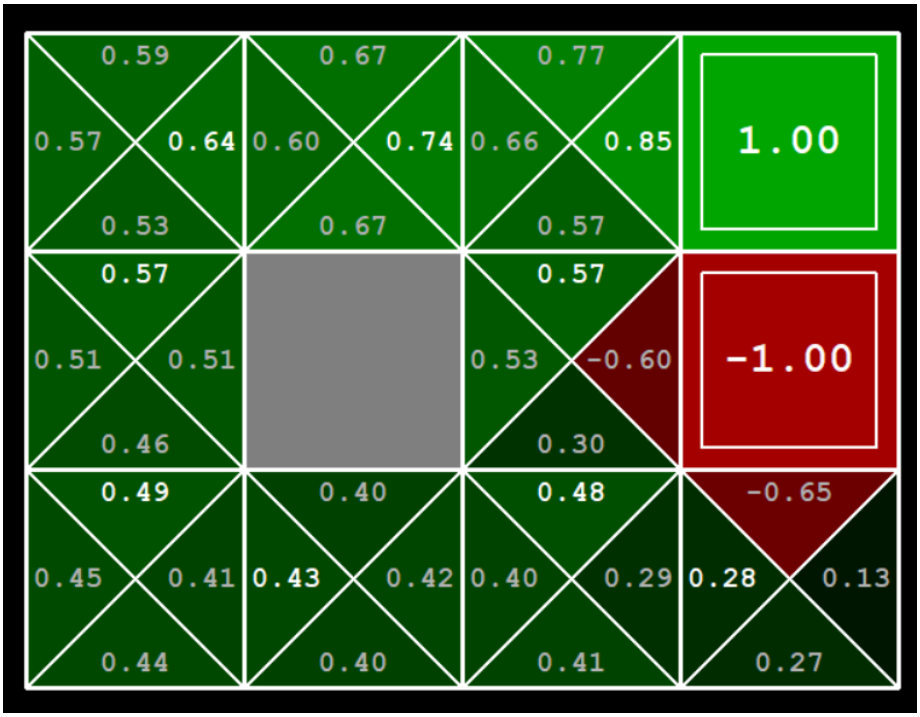

图2 迭代101次

为了实现上述代码，我学习了YouTube的视频以及github上的代码，才真正理解值迭代的算法，并还将学习的过程记载博客中。

Project 2 桥梁问题分析

Project1中使用的是默认的GAMMA(0.9)以及Noise(0.3)，但在这种情况下会导致在处理桥梁问题时，由于过于谨慎，而不敢前进。如下图所示：

图3 不敢在桥上走很久

导致这样的原因是因为AI的行为过于不确定，每走一步都会有30%的几率走错路，这种行为在吊桥上是十分危险的，因此会趋向于保守地快速从吊桥走出去。为此我修改了Noise值，使其变成0.01。这样每一步操作都会有99%是按照预期来行动的，这样掉下去的概率就会大大降低，因此AI就可以顺利走到奖励值大的地方了。

图4 顺利抵达终点

Project 3 更多的行动策略

在一张更为复杂的地图中，想要实现不同策略耶更加复杂，如图5

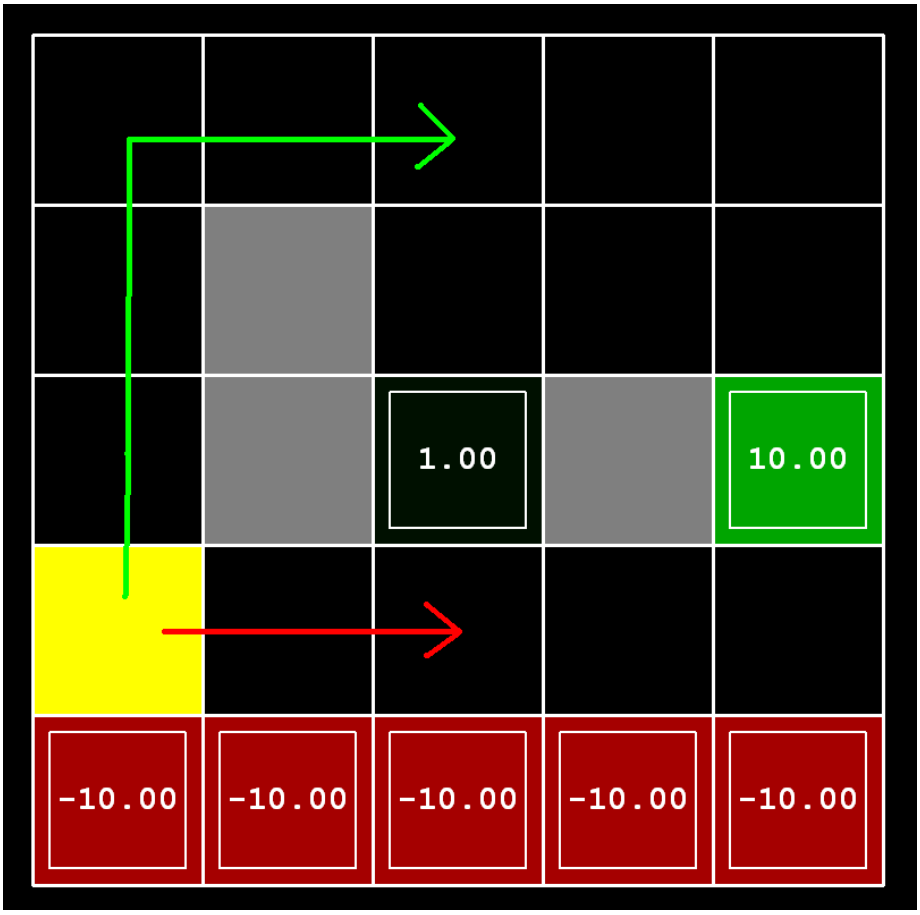

图5 复杂地图

修改`answerDiscount,answerNoise,answerLivingReward`实现不同策略：

贴着悬崖，到达近的出口

为了贴着悬崖，需要更小的Noise，为了能从近的出口出去，需要更大的存活惩罚，逼迫智能体尽快找到一个出口逃离。因此最终的参数选择为

- answerDiscount = 0.9
- answerNoise = 0.1
- answerLivingReward = -4

绕开悬崖，到达近的出口

这个我没想出来...参考了别人的赋值：

- answerDiscount = 0.1
- answerNoise = 0.1
- answerLivingReward = -3

事后诸葛亮的分析一下，可以看到节点之间的相互影响更小，并且惩罚值很大，促使AI快速找到终态。

贴着悬崖，到达远的出口

与第一个同理，要减小Noise，不过为了能到达更远(并且奖励更大)的出口，需要减少存活的惩罚值，可以让智能体更多的游荡一会儿，并找到奖励更大的路线。

- answerDiscount = 0.9
- answerNoise = 0.1
- answerLivingReward = -1

绕开悬崖，到达远的出口

为了绕开悬崖，可以提升Noise，让在悬崖边的状态更加危险。同样降低惩罚值，让AI闲逛找到最远的出口。

- answerDiscount = 0.9
- answerNoise = 0.3
- answerLivingReward = -1

永远到达不了出口的真实

最后我将存活的奖励值调大，只要一直处于闲逛状态就能获得巨额分数，那么谁还愿意从出口出去呢？

- answerDiscount = 0.9
- answerNoise = 0.3
- answerLivingReward = 100

总结

我认为收获最大的是Project 1的算法，让我有种实在掌握知识的感觉。Project 2和3多少有些算命的感觉，即使做出来，成就感也不是很强。这个实习从简单的例子开始讲起，让我更直观的了解了强化学习中值迭代的MDP使用过程，受益匪浅。

111171 董安宁