

实验指导书

任务1 基于动态规划的路径规划

冰湖路径规划

目 录

[一、实验信息 1](#_Toc24145)

[二、实验准备 1](#_Toc15244)

[1. 实验环境 1](#_Toc29427)

[2. 实验数据 4](#_Toc24936)

[三、实验步骤 4](#_Toc4671)

[1. 计算状态价值函数 4](#_Toc7306)

[2. 计算动作价值函数 6](#_Toc13035)

[3. 策略迭代 8](#_Toc27888)

[4. 价值迭代 10](#_Toc12439)

[四、实验总结 13](#_Toc5357)

[五、实验思考 13](#_Toc3253)

[六、课后任务 14](#_Toc24306)

一、实验信息

|  |  |
| --- | --- |
| 实验名称 | 基于动态规划的路径规划 |
| 实验等级 | 中阶 |
| 实验时长 | 2课时 |
| 实验内容 | 1. 计算状态价值函数 2. 计算动作价值函数 3. 实现策略迭代算法 4. 实现价值迭代算法 |
| 实验目标 | 1. 理解动态规划的基本思想  2. 掌握利用贝尔曼方程评估价值函数  3. 掌握策略迭代算法  4. 掌握价值迭代算法 |

二、实验准备

### 1. 实验环境

|  |  |  |
| --- | --- | --- |
| 序列 | 名称 | 规格/版本 |
| 硬件 | PC计算机或服务器 | 1G以上内存，40G 以上硬盘，有英伟达GPU更佳 |
| 开发语言 | python | 3.8.13 |
| 强化学习环境 | gym | 0.24.1 |
| 数值计算 | numpy | 1.23.0 |

#### 1.1 环境安装

本次实验将基于OpenAI的Gym环境。OpenAI是一家非营利性的人工智能研究公司，公布了非常多的学习资源以及算法资源。其之所以叫作OpenAI，是因为他们把所有开发的算法都进行了开源。OpenAI 的 Gym库是一个环境仿真库，里面包含很多现有的环境。针对不同的场景，我们可以选择不同的环境。

为了下载并安装 OpenAI Gym，我们可以使用以下方法：

|  |
| --- |
| pip install gym # minimal install  pip install gym[all] # full install  pip install gym[atari] # for Atari specific environment installation |

#### 1.2 环境介绍

在本实验中将会用到Gym中的冰湖环境（Frozen Lake），冰湖环境是一个网格环境，大小为 ，见图x-x。每一个方格对应了一个状态，智能体起点状态在左上角，目标状态在右下角，中间还有若干冰洞。在每一个状态都可以采取上、下、左、右 4 个动作。由于智能体在冰面行走，因此每次行走都有一定的概率滑行到附近的其它状态。当掉入冰洞或到达目标状态时结束。每一步行走的奖励是 0，到达目标的奖励是 1。



图x-x 冰湖环境

我们可以通过如下代码查看一条随机轨迹：

|  |
| --- |
| import gym  env = gym.make("FrozenLake-v1") # 创建环境  env.reset()  for t in range(100):  env.render() # 渲染画面  a = env.action\_space.sample() # 随机采样动作  observation, reward, done, \_ = env.step(a) # 环境执行动作，获得转移后的状态、奖励以及环境是否终止的指示  if done:  break  env.render() |

我们可以通过如下代码获取该环境的状态转移函数与奖励函数：

|  |
| --- |
| env = env.unwrapped  P = env.P |

P为一个字典，对于每一个状态动作对（state, action），P[state][action]返回一个形如（probability, next\_state, reward, terminal）的元组，其中：

* + probability为在状态“state”采取动作“action”转移到状态“next\_state”的概率。
  + next\_state为转移到的下一个状态。
  + reward为发生状态转移所获得的奖励。
  + terminal用于指示next\_state是否为终止状态。

为了方便本次实验，我们可以将其转化为如下所示的MDP类：

|  |
| --- |
| class MDP(object):  def \_\_init\_\_(self, P, nS, nA):  self.P = P # state transition and reward probabilities, explained below  self.nS = nS # number of states  self.nA = nA # number of actions |

可以通过下面代码获得一个MDP类的实例对象。

|  |
| --- |
| mdp = MDP( {s : {a : [tup[:3] for tup in tups] for (a, tups) in a2d.items()} for (s, a2d) in env.P.items()}, env.observation\_space.n, env.action\_space.n) |

### 实验数据

无。

三、实验步骤

### 1. 计算状态价值函数

【操作介绍】策略评估过程用于计算一个策略的状态价值函数，回顾之前学习的贝尔曼期望方程:

其中， 是策略 在状态 下采取动作 的概率。可以看到，当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。因此，根据动态规划的思想，可以把计算下一个可能状态的价值当成一个子问题，把计算当前状态的价值看作当前问题。在得知子问题的解后，就可以求解当前问题。更一般的，考虑所有的状态，就变成了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:

我们可以设定任意初始值 。根据贝尔曼期望方程，可以得知 是以上更新公式的一个不动点（fixed point）。事实上，可以证明当 时，序列 会收敛到 ，所以可以据此来计算得到一个策略的状态价值函数。在实际的实现过程中，如果某一轮 的值非常小了，就可以提前结束策略评估。

具体而言，我们在本次任务需要实现compute\_vpi函数，用于计算获得马尔可夫决策过程中任意策略 的状态价值函数 。

#### 1.1 计算状态价值函数

|  |
| --- |
| def compute\_vpi(pi, mdp, gamma):  # pi: 用于评估的确定性策略，用列表进行表示，其维度与状态数相同，用于输出每个状态下所要采取的确定性动作。  # mdp: 环境对应的MDP类实例。  # gamma: 奖励折扣因子。  # 同学们需要实现该函数，返回状态价值函数V，V使用numpy数组进行表示，维度与状态数相同。  return V |

#### 1.2 compute\_vpi函数运行样例

|  |
| --- |
| print(compute\_vpi(np.ones(16), mdp, gamma=GAMMA))  输出如下：  [ 1.638e-02 2.357e-02 2.317e-01 2.433e-02 1.656e-02 -0.000e+00  2.989e-01 0.000e+00 1.972e-02 1.879e-01 3.934e-01 0.000e+00  -3.581e-17 1.956e-01 4.941e-01 0.000e+00] |

### 2. 计算动作价值函数

使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。假设此时对于策略 ，我们已经知道其价值 ，也就是知道了在策略下从每一个状态出发最终得到的期望回报。我们要如何改变策略来获得在状态 下更高的期望回报呢？假设智能体在状态 下采取动作 ，之后的动作依旧遵循策略 ，此时得到的期望回报其实就是策略 下的动作价值 。如果此时满足 ，则说明在状态 下采取动作 会比原来的策略 得到更高的期望回报。以上假设只是针对一个状态，现在假设存在一个确定性策略 ，在任意一个状态下，都满足:

于是在任意状态下，我们有,

因此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，获得一个新的策略，也就是，

我们可以知道，策略 π′ 比策略 π 更好或至少一样好。

#### 2.1实现compute\_qpi函数并根据状态价值函数与状态转移概率计算获得动作价值函数。

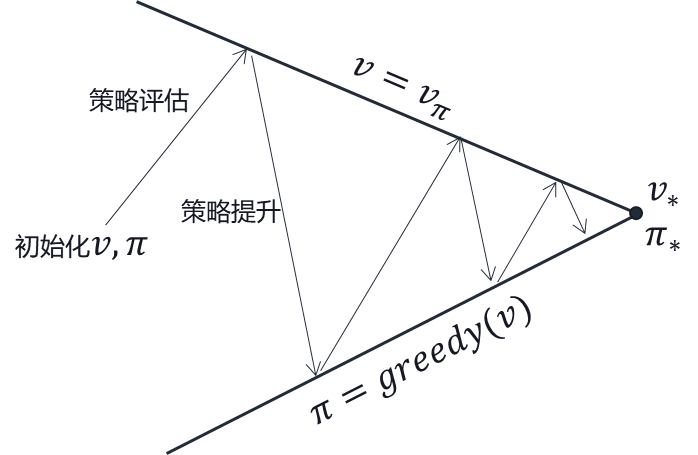
|  |
| --- |
| def compute\_qpi(vpi, mdp, gamma):  # vpi: 状态价值函数，使用列表表示，维度与状态数相同。  # mdp: 环境对应的MDP类实例。  # gamma: 奖励折扣因子。  # 同学们需要实现该函数，返回动作价值函数。  return Qpi |

#### 2.2 compute\_qpi函数运行样例

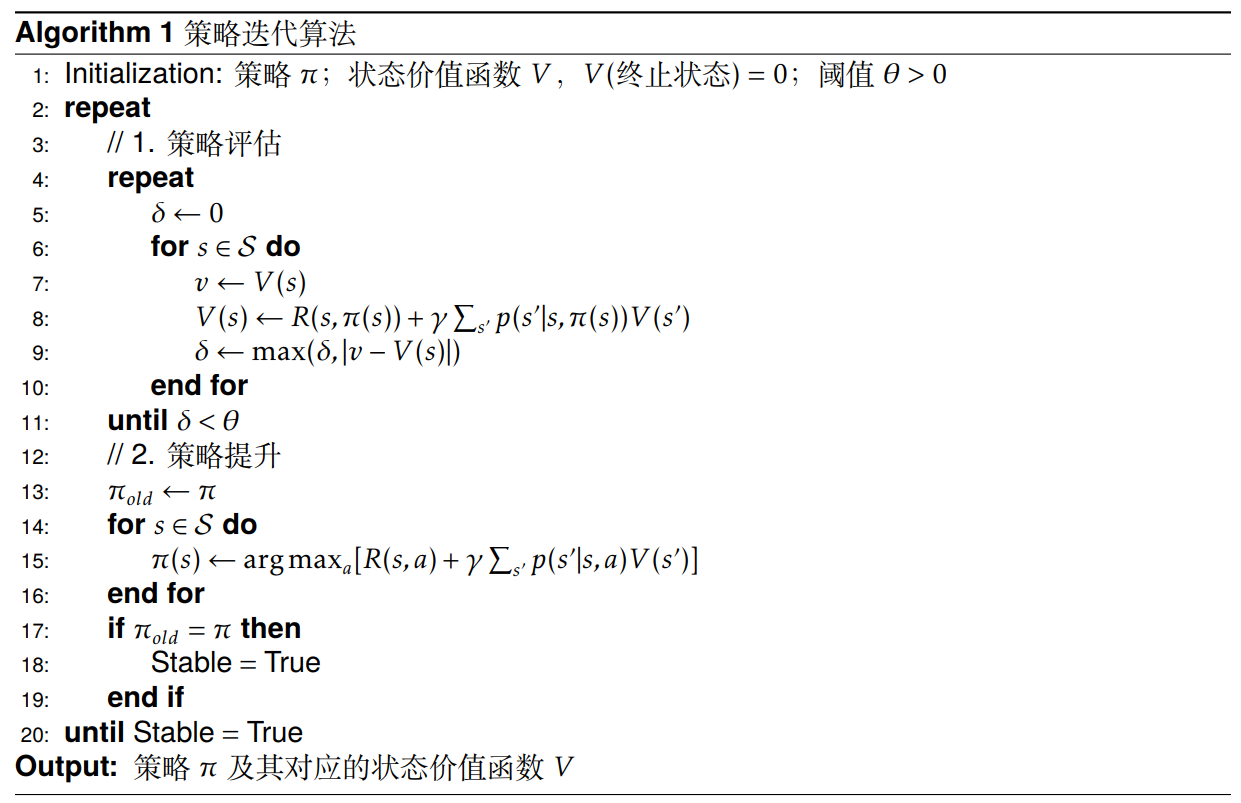
|  |
| --- |
| Qpi = compute\_qpi(np.arange(mdp.nS), mdp, gamma=0.95)  print("Qpi:\n", Qpi)  输出如下：  Qpi:  [[ 0.38 3.135 1.14 0.095]  [ 0.57 3.99 2.09 0.95 ]  [ 1.52 4.94 3.04 1.9 ]  [ 2.47 5.795 3.23 2.755]  [ 3.8 6.935 4.56 0.855]  [ 4.75 4.75 4.75 4.75 ]  [ 4.94 8.74 6.46 2.66 ]  [ 6.65 6.65 6.65 6.65 ]  [ 7.6 10.735 8.36 4.655]  [ 7.79 11.59 9.31 5.51 ]  [ 8.74 12.54 10.26 6.46 ]  [ 10.45 10.45 10.45 10.45 ]  [ 11.4 11.4 11.4 11.4 ]  [ 11.21 12.35 12.73 9.31 ]  [ 12.16 13.4 14.48 10.36 ]  [ 14.25 14.25 14.25 14.25 ]] |

### 3. 策略迭代

总体来说，策略迭代算法的过程如下：对当前的策略进行策略评估，得到其状态价值函数，然后根据该状态价值函数进行策略提升以得到一个更好的新策略，接着继续评估新策略、提升策略……直至最后收敛到最优策略：



结合策略评估与策略提升，我们可以得到策略迭代算法：



#### 3.1 实现策略迭代

在实现compute\_vpi函数与compute\_qpi函数后，我们可以利用这两个函数实现策略迭代算法，如下所示。

|  |
| --- |
| import numpy as np  def policy\_iteration(mdp, gamma, nIt):  # mdp: 环境对应的MDP类实例。  # gamma: 奖励折扣因子。  # nlt: 迭代次数。  Vs = []  pis = []  pi\_prev = np.zeros(mdp.nS,dtype='int')  pis.append(pi\_prev)  print("Iteration | # chg actions | V[0]")  print("----------+---------------+---------")  for it in range(nIt):  vpi = compute\_vpi(pi\_prev, mdp, gamma)  qpi = compute\_qpi(vpi, mdp, gamma)  pi = qpi.argmax(axis=1)  print("%4i | %6i | %6.5f"%(it, (pi != pi\_prev).sum(), vpi[0]))  Vs.append(vpi)  pis.append(pi)  pi\_prev = pi  return Vs, pis  Vs\_PI, pis\_PI = policy\_iteration(mdp, gamma=0.95, nIt=20)  plt.plot(Vs\_PI) |

### 4. 价值迭代

策略迭代中的策略评估需要进行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，尤其是在状态和动作空间比较大的情况下。我们是否必须要完全等到策略评估完成后再进行策略提升呢？试想一下，可能出现这样的情况：虽然状态价值函数还没有收敛，但是不论接下来怎么更新状态价值，策略提升得到的都是同一个策略。如果只在策略评估中进行一轮价值更新，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，这其实就是本节将要讲解的价值迭代算法，它可以被认为是一种策略评估只进行了一轮更新的策略迭代算法。需要注意的是，价值迭代中不存在显式的策略，我们只维护一个状态价值函数。

#### 4.1 实现value\_iteration函数

|  |
| --- |
| def value\_iteration(mdp, gamma, nIt):  """  输入:  mdp: 环境对应的MDP类实例  gamma: 奖励折扣因子  nIt: 迭代次数  输出:  (value\_functions, policies)    其中len(value\_functions) == nIt+1 and len(policies) == n  """  print("Iteration | max|V-Vprev| | # change actions | V[0]")  print("----------+--------------+---------------+---------")  Vs = [np.zeros(mdp.nS)] # 价值函数列表，初始化为0  pis = [] # 历史策略列表  for it in range(nIt):  oldpi = pis[-1] if len(pis) > 0 else None  Vprev = Vs[-1] # V^{(it)}  # 以下是需要实现的代码  # 你的代码需要定义如下两个变量  # pi: 基于价值函数Vprev的贪心策略, 为一个整型numpy数组  # V: 在Vprev基础上进行贝尔曼备份，为一个浮点型numpy数组  max\_diff = np.abs(V - Vprev).max()  nChgActions="N/A" if oldpi is None else (pi != oldpi).sum()  print("%4i | %6.5f | %4s | %5.3f"%(it, max\_diff, nChgActions, V[0]))  Vs.append(V)  pis.append(pi)  return Vs, pis |

#### 4.2 策略测试

|  |
| --- |
| def test\_pi(env, pi, num\_episodes=100):  """  测试策略。  参数：  env -- OpenAI Gym环境对象。  pi -- 需要测试的策略。  num\_episodes -- 进行测试的回合数。  返回值：  成功到达终点的频率。  """  count = 0  for e in range(num\_episodes):  ob = env.reset()  for t in range(100):  a = pi[ob]  ob, rew, done, \_ = env.step(a)  if done:  count += 1 if rew == 1 else 0  break  return count / num\_episodes  # pi = policy\_iteration\_MC(env, 1, num\_episodes=10000)  # result = test\_pi(env, optimal\_pi)  # print(result) |

四、实验总结

1. 本次实验基于Gym的冰湖环境实现了两种基于动态规划的最优策略求解算法——策略迭代与价值迭代。
2. 策略迭代包括策略评估与策略提升两个阶段，策略评估阶段求解策略 下的价值函数，策略提升阶段依据价值函数导出一个贪心策略。
3. 策略迭代中的策略评估收敛所需耗时较大，而在实际过程中评估无需收敛也可进行策略提升。如果只在策略评估中进行一轮价值更新，然后直接根据更新后的价值进行策略提升，这就是价值迭代算法。

五、实验思考

问题1. 如果数据流不具备马尔可夫性质怎么办？应该如何处理？

问题2. 目前介绍的动态规划算法每次都需扫描所有状态进行更新，计算复杂度较高，有何优化方法？

六、课后任务

1. 尝试实现异步动态规划算法，对比与同步动态规划的区别。

2. 思考当冰湖环境中的状态转移不仅与当前状态与动作有关，还依赖于前一状态时，该如何处理？