一、实验内容：

作业 1：马尔可夫过程价值函数计算
作业 2：基于动态规划的强化学习算法实现
作业 3：Sarsa 和 Q-learning 算法实现

二、实验要求：

（一）实验环境：https://gymnasium.farama.org/environments/toy_text/frozen_lake/
（二）代码要求：程序要添加适当的注释，代码命名规范，可读性高。
（三）实验报告：表达清楚，文字准确、可图文并茂。重点在于算法总结，逻辑清晰，语言精炼，将碰到的问题、解决方法、结果分析、讨论与思考记录下来。最终上交 pdf 文件，格式自拟，不要粘贴源代码。
（四）提交材料：将实验代码、实验报告打包成 zip 压缩包，命名格式为**学号-姓名.zip**，例如 **2023000100-张三.zip**。
（五）提交方式：将 zip 压缩文件提交到教学云平台。

三、实现方法、实验结果及结论分析

1.实现方法及算法总结

1.1 马尔可夫过程价值函数计算

计算下图各个状态的值按照贝尔曼方程代入数据即可

贝尔曼方程：

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

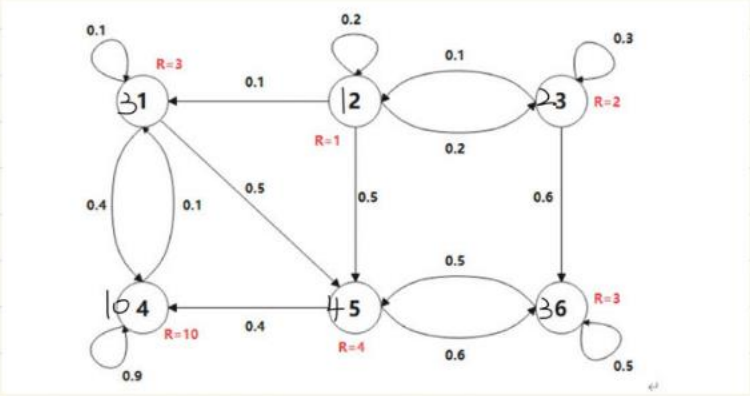
注：更新方式使用的是同步更新方式，即计算完所有状态的价值之后同步进行更新再进入下一次迭代过程

马尔夫奖励过程 由元组 (S, P, R_s, γ) 构成，状态集合 $S=\{S1, S2, S3, S4, S5, S6\}$

初始时 $v(s)=E[G_t|S_t=s]=0$，衰减因子 $\gamma=0.9$，用 s' 表示状态，s 的下一时刻作为继续状态，则贝尔曼方程表示为

$$v(s)=R_s+\gamma\sum_{s\in S}P_{ss'}v(s')$$

其中 $P_{ss'}=P[S_{t+1}=s'|S_t=s]$，$R_s=E[R_{t+1}|S_t=s]$



第一次迭代：（遍历所有 $S\in S$）

$v(1)=3+0.9\times(0.1\times0+0.4\times0+0.5\times0)=3$

$v(2)=1+0.9\times(0.5\times0+0.2\times0+0.2\times0+0.1\times0)=1$

$v(3)=2+0.9\times(0.3\times0+0.6\times0+0.1\times0)=2$

$v(4)=10+0.9\times(0.9\times0+0.1\times0)=10$

$v(5)=4+0.9\times(0.6\times0+0.4\times0)=4$

$v(6)=3+0.9\times(0.5\times0+0.5\times0)=3$

第二次迭代：

$v(1)=3+0.9\times(0.1\times3+0.5\times4+0.4\times10)=8.67$

$v(2)=1+0.9\times(0.2\times1+0.1\times3+0.2\times2+0.5\times4)=3.61$

$v(3)=2+0.9\times(0.3\times2+0.6\times3+0.1\times1)=4.25$

$v(4)=10+0.9\times(0.9\times10+0.1\times3)=18.37$

$v(5)=4+0.9\times(0.4\times10+0.6\times3)=8.22$

$v(6)=3+0.9\times(0.5\times3+0.5\times4)=6.15$

1.2 基于动态规划的强化学习算法

在开始之前简单介绍使用的环境 Frozen Lake，FrozenLake 是典型的具有离散状态空间的 Gym 环境，agent 的任务是学会从起点走到目的地并不能掉进 hole。

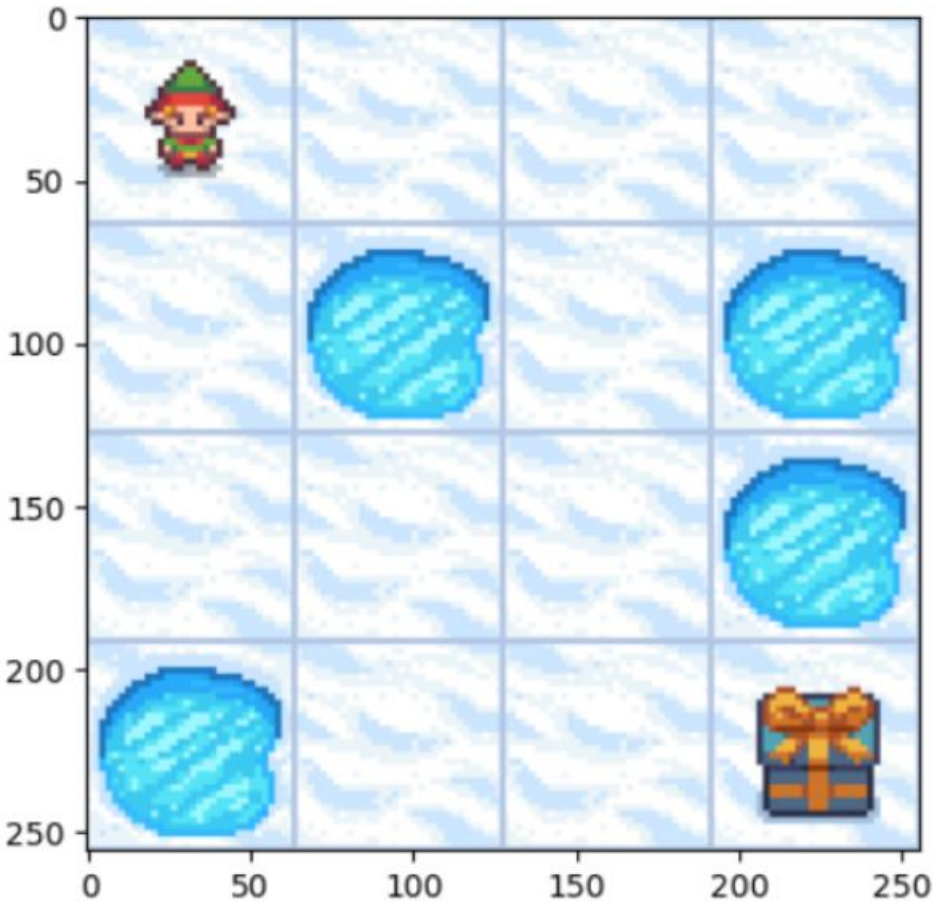
Frozen Lake 游戏中，一块冰面上有四种 state：

- S: initial stat 起点
- F: frozen lake 冰湖
- H: hole 窟窿，可用于结束一个回合
- G: the goal 目的地，可用于结束一个回合

利用 matplotlib 可以查看冰冻湖的原始图像

```python
import gymnasium as gym
import matplotlib.pyplot as plt

env = gym.make('FrozenLake-v1', desc=["SFFF", "FHFH", "FFFH", "HFFG"], map_name="4x4", is_slippery=True,render_mode='rgb_array')

# Reset the environment to a random initial state
obs = env.reset()

# Render the environment as an RGB array
img = env.render()

# Plot the RGB image using matplotlib
plt.imshow(img)
plt.show()
```

绘制的图像如下



可将上述冰冻湖转换为如下图例

SFFF
FHFH
FFFH
HFFG

冰冻湖的状态个数为 16(0-15)，行为个数为 4(LEFT=0；DOWN=1；RIGHT=2；UP=3)，需要注意的是，在 FrozenLake 环境中，由于冰面很滑，因此智能体不会始终按照指定的方向移动。例如，当执行向下移动的动作时，智能体也可能向左或向右移动（当然可以在后续将 is_slipper 关闭，则动作的随机性消失）。

如果满足以下两个条件之一，则回合将终止：

- 智能体移动到 H 格子(状态 5、7、11 或 12)：产生的总奖励为 0
- 智能体移动到 G 格子(状态 15)：产生的总奖励为 +1

Frozen Lake 的源码中的 step 部分如下（这是 Gym 中的 step，Gymnasium 中的 step 中间的 done 被拆分为 terminated 和 truncated 返回）

```
def step(self, a):
    transitions = self.P[self.s][a]
    i = categorical_sample([t[0] for t in transitions], self.np_random)
    p, s, r, d = transitions[i]
    self.s = s
    self.lastaction = a
    return (int(s), r, d, {"prob": p})
```

即 step 部分接受动作 a 并返回状态 s，奖励 r，是否接收 d 以及转移概率 p，其中转移概率的通俗解释为“在当前的状态，如果动作向左，就会有向着除了反方向之外的格子 1/3 的概率”

```
For example, if action is left and is_slippery is True, then:
        - P(move left)=1/3
        - P(move up)=1/3
        - P(move down)=1/3
```

实验的目标是设计算法使 agent 能够学习得到最好的策略，即在每个状态选择最优的动作，最终能够到达终点。

在开始之前我们不妨让 agent 选择随机策略，即在每个状态随机从(0,3)之间随机选取一个数字作为 action_index 执行动作，看看效果怎么样（返回的矩阵表示 agent 在冰冻湖的位置，返回的元组第一个参数表示 next_state，第二个参数表示 rewards，这个参数只有在达到终点的时候才是 1，第三个参数表示事件是否结束）

```
count = 0
while True:
    count = count+1
    action=random.randint(0,3)
    env.print_current_state()
#     time.sleep(1)
    new_observation=env.step(action) #这里的接收的new_observation元组分别是当前状态、奖励以及指示事件是否结束
    print(new_observation)
    print('\n')
    if new_observation[2]: #如果事件结束则reset状态（不管奖励是否>1，先判定事件是否结束）
        env.reset()
    if new_observation[1]>0: #如果奖励>0 ,也就是到达目的地，则退出循环
        print('Reach goals,try '+str(count)+'times')
        break
```

随即策略因为是 agent 随机选择的策略，所以就算我们的 action 是确定的，但是 agent 仍然需要执行很多次的尝试

```
S F F F
F H F H
F F F H
H F X G
(14, 0, False)


S F F F
F H F H
F F F H
H F X G
(15, 1, True)


Reach goals, try 1306times
```

可见使用随机策略不是一个可行的方式，下面将分别使用动态规划的方法和无模型控制的方法为 agent 寻找最优策略。

1.2.1 策略迭代

(1)算法介绍

策略迭代主要分为策略评估、策略提升和策略迭代三部分。

策略评估顾名思义就是评估谁的策略 Π 更好，经过前面的学习知道可以将该问题转化，即预测一个给定的策略 Π 在所有状态 s 下的价值函数 v_Π。

如何求出给定策略 Π 在所有状态 s 下的价值函数呢？ -- 迭代使用贝尔曼期望方程进行回溯，用初始价值函数迭代更新后继价值函数，直到算法收敛到 v_Π，因此该方法也称为迭代策略评估

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\pi$$

初始价值函数　　收敛价值函数

$$\forall s:\ v_{k+1}(s) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s]$$

下面给出策略评估的算法伪代码：

循环：

$\Delta \leftarrow 0$

对每一个 $s \in S$ 循环：

$v \leftarrow V(s)$

$$V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V(s') \right)$$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

直到 $\Delta < \theta$ （一个确定估计量的精度小正数）

- 系统任意初始化得到的 V(s)并不是该策略 Π 真正的价值函数，因为并不满足贝尔曼期望方程，只有整个系统收敛时得到的价值函数才会满足贝尔曼期望方程；
- 系统收敛得到的 v_Π 一定满足贝尔曼期望方程，但是不一定满足最优方程，因为只有最优策略的 v_Π 才满足贝尔曼最优方程；

前面介绍了如何预测一个给定策略Π在所有状态 s 下的价值函数 v~Π~，是否可以根据该价值函数来改进该策略得到策略Π'，使得策略Π'的价值函数高于策略Π的价值函数？

这里我们应用贪婪方法来改进策略Π，具体使用的是当时最大化动作价值函数寻找最优策略的方式（实际上就是调整策略Π在状态 s 下对应的动作 a 的概率）。

$$\pi_*(a|s) = \begin{cases} 1 & 当 a = arg\max_{a \in A} q_*(s,a) \\ 0 & \end{cases}$$

策略提升的伪代码如下

$Policy - stable \leftarrow true$

对每一个 $s \in S$：

$old - action \leftarrow \pi(s)$

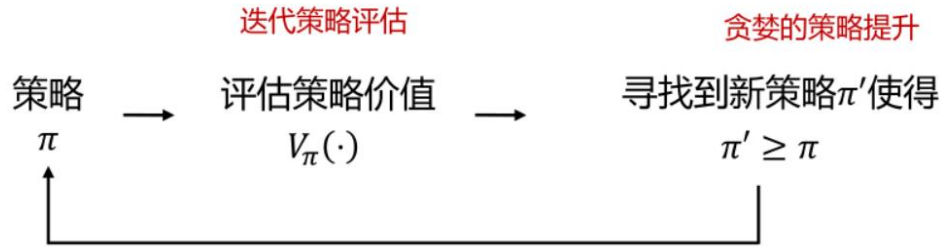
$$\pi(s) \leftarrow arg\max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V(s') \right)$$

如果 $old - action \neq \pi(s)$，那么 $Policy - stable \leftarrow false$

如果 $Policy - stable$ 为 $true$，那么停止并返回 $V = v_*$ 以及 $\pi = \pi_*$；

否则跳转到 2

在大部分问题中，需要进行多次的策略评估和策略改进的迭代才能得到该问题的最优策略



(2)算法实现

如伪代码所示，策略迭代算法主要分为三个部分：策略评估、策略提升以及策略迭代，我们分别介绍。

首先是策略评估，主要思想是计算当前策略 pi 的状态价值函数 v，并通过迭代所有状态，计算每个状态-操作对的期望值来评估当前策略。

- 策略评估算法首先将所有状态的值函数 v 初始化为零；

- 在环境中的所有状态上迭代，对于每个状态，在所有操作上迭代，以计算每个操作的预期回报 qsa；预期回报是通过将所有可能的下一个状态的奖励和 discounted 的未来收益求和，并按其概率加权来计算；

- 循环持续直到所有状态的新的和旧的值函数之间的最大差值小于给定的阈值 θ；

- 最后使用更新后的值函数来评估策略；

```
def policy_evaluation(self):
    while 1:
        delta = 0 # 每次迭代开始时初始化为0，以跟踪值函数的最大变化
        new_v = [0] * self.env.ncol * self.env.nrow # 将状态值函数v初始化为零
        for s in range(self.env.ncol * self.env.nrow): # 在环境中所有状态上循环，实现在状态空间列表上迭代
            exp_list = []
            for a in range(4): # 对于每个状态s，迭代所有动作来计算每个状态的状态值函数
                exp = 0 # 用于跟踪状态s中每个动作a的预期回报
                for res in self.env.P[s][a]: # 预期回报是通过在状态s中对采取行动a的所有可能结果进行循环
                    # 预期回报=下一状态next_state的奖励和discounted value之和
                    exp += res[0] * (res[2] + self.gamma * self.V[res[1]]*(1-res[3]))
                # 将所有操作的加权预期回报相加来计算状态s的新值
                exp_list.append(self.policy[s][a] * exp)
            new_v[s] = sum(exp_list)
            # 计算新的和旧的状态值函数之间的最大差值
            delta = max(delta, abs(new_v[s] - self.V[s]))
        # self.v变量被更新为新的值函数new_v
        self.V = copy.deepcopy(new_v)
        if delta < self.theta:
            break
    print('Policy Evaluation Done!')
```

接着是策略提升，策略提升函数的输入为当前状态值函数 self.v，更新策略 self.pi，简单来说策略提升的目标是更新 agent 的策略以选择每个状态的期望积累奖励最大化的操作。

- 策略提升对环境中所有可能的状态进行迭代，对每个状态使用贝尔曼方程计算每个可能 action 的期望回报，每个 action 的期望返回存储在该状态对应的 q 值列表中；
- 选择具有最高 q 值的 action，同时将选择这些 action 中的每一个的概率设置为 1 除以具有最高 q 值的动作的数量，其他 action 概率设置为 0；
- 函数返回更新后的 policy，用于之后的策略评估和策略提升；

```
    def policy_improvement(self):# 策略提升，agent通过选择使每个状态的预期discounted累积奖励最大化的操作来更新其策略
        for s in range(self.env.ncol * self.env.nrow):# 在环境中所有可能的状态s上循环
            exp_list = []
            for a in range(4):
                exp = 0# 对于每个状态s，agent计算每个可能动作的预期siacounted积累奖励
                for res in self.env.P[s][a]:
                    # 贝尔曼方程 qsa = sum(p * (r + gamma * v(next_state)) for p, next_state, r, done in P[s][a])
                    # P[s][a]是一组可能的下一个状态、奖励和转换到下一个状态的概率
                    exp += res[0] * (res[2] + self.gamma * self.V[res[1]]*(1-res[3]))
                exp_list.append(exp)
            max_exp = max(exp_list)# 选择最大值maxq并计算具有该最大值的动作数量cntq
            cnt_max = exp_list.count(max_exp)
            # 所有具有最大动作值的动作被选中的概率相等，而所有其他动作的概率为0
            self.policy[s] = [1 / cnt_max if q == max_exp else 0 for q in exp_list]
        print('Policy Improvement Done!')
        return self.policy
```

最后是策略迭代部分，策略迭代指的是 agent 与 env 交互学习得到 policy，使得期望的积累回报最大化；

- 策略迭代函数通过交替调用策略评估和策略提升算法，直至策略收敛，收敛指的是当策略在两个连续的迭代之间不再改变即 old_policy=new_policy；

```
    def policy_iteration(self):
        while 1:
            self.policy_evaluation() # 调用policy_evaluation函数，通过计算当前策略的状态值函数v来评估该策略
            old_policy = copy.deepcopy(self.policy)# 将列表进行深拷贝，方便接下来进行比较
            new_policy = self.policy_improvement()# 调用policy_improvement函数，通过选择最大化预期长期回报的行动（即行动值函数q）来改进当前策略
            if old_policy == new_policy:
                break
        print('Policy Iteration Done!')
```

(3)算法实践

首先执行 train 函数，进入训练阶段，也就是不断进行策略迭代

```
# 策略迭代
print('PolicyIteration Beginning...')
agent = PolicyIterationAgent(env, theta, gamma)
agent.policy_iteration()
```

训练完成得到如下输出

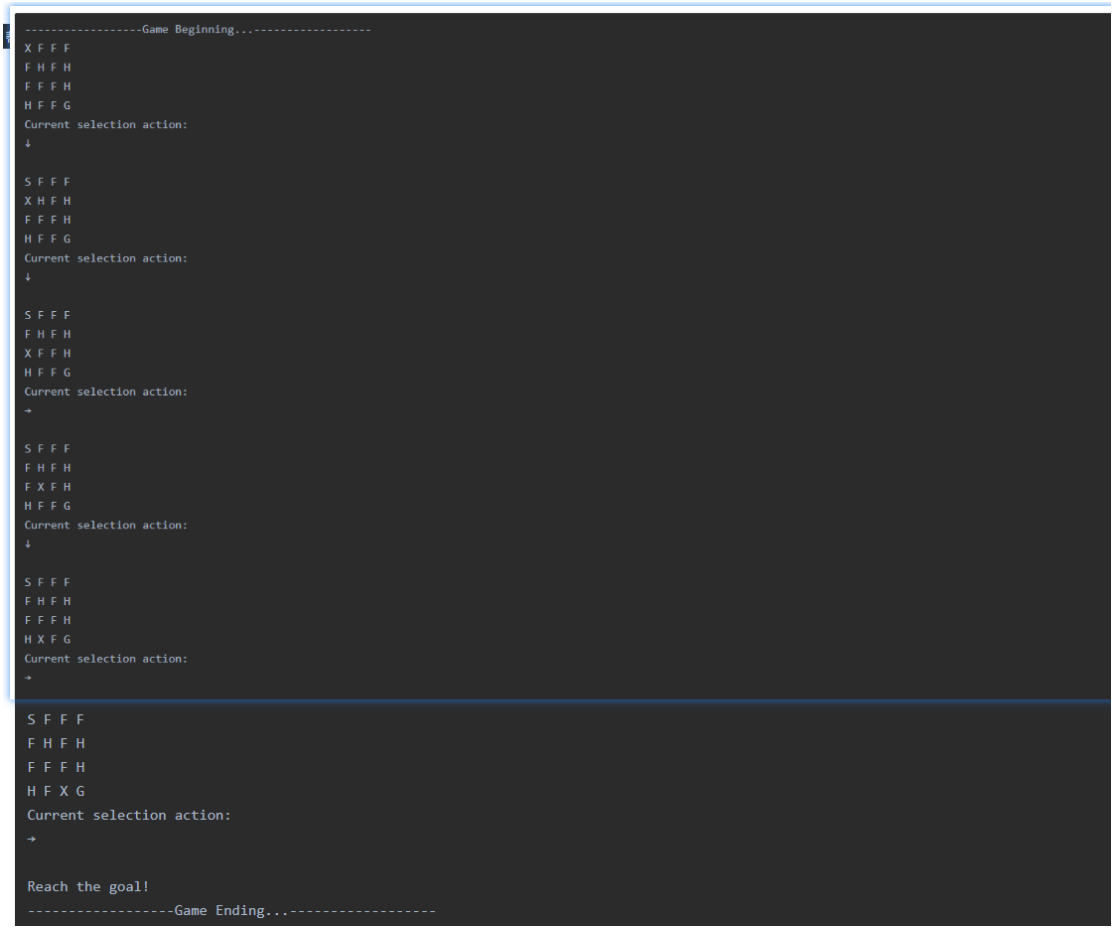
```
PolicyIteration Beginning...
Policy Evaluation Done!
Policy Improvement Done!
Policy Evaluation Done!
Policy Improvement Done!
Policy Evaluation Done!
Policy Improvement Done!
Policy Iteration Done!
------------------OptimalValue------------------
 0.590  0.656  0.729  0.656
 0.656  0.000  0.810  0.000
 0.729  0.810  0.900  0.000
 0.000  0.900  1.000  0.000
------------------OptimalPolicy------------------
State 0 -> Optimal Action:↓
State 0 -> Optimal Action:→
State 1 -> Optimal Action:→
State 2 -> Optimal Action:↓
State 3 -> Optimal Action:←
State 4 -> Optimal Action:↓
State 5 -> Optimal Action: Hole
State 6 -> Optimal Action:↓
State 7 -> Optimal Action: Hole
State 8 -> Optimal Action:→
State 9 -> Optimal Action:↓
State 9 -> Optimal Action:→
State 10 -> Optimal Action:↓
State 11 -> Optimal Action: Hole
State 12 -> Optimal Action: Hole
State 13 -> Optimal Action:→
State 14 -> Optimal Action:→
State 15 -> Optimal Action: End
```

可以看到在动作无随机性的情况下策略迭代进行了三轮，即交替执行三次策略评估和策略提升，最终得到每个状态对应的最优价值。

最优策略显示了在某个状态时可以选择执行的 action，推导的原则为若邻居状态的价值大于当前状态的价值函数则向邻居状态移动，因此也可以看到有些状态如 0 既可以向下移动也可

以向右移动，因为这两个方向的价值函数均大于 0 状态本身的价值函数。

拥有了最优策略我们就可以将该策略给 agent，指导 agent 成功的从起点移动到终点，下面执行 test 函数（test 函数的原理非常简单，就是根据 agent 的 policy 在每个状态选择 action 并输出可视化结果）对得到的最优策略进行检验，得到如下输出



X 表示 agent 当前所在位置，Current selection action 表示 agent 在当前状态选择的 action，依据是前面得到的最优策略（如果某个状态对应两个动作则随机从中选取一个动作）。

可以很直观的从图中看到，agent 成功的从起点到达终点，并且成功的避免了掉进冰窟中或移动到 grid 外面。

1.2.2 价值迭代

(1)算法介绍
在策略迭代的策略评估过程中，我们并不需要迭代到 k=∞即系统收敛时才进行策略提升，当策略评估迭代的次数 k 越小表示运算量越小，当 k=1 时，此时的策略迭代就成了价值迭代在价值迭代中，因为只需要进行一次策略评估

对每一个 $s \in S$ 循环：

$$V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V(s') \right)$$

而在策略提升中会使用最大化动作价值函数

$$\pi(s) \leftarrow arg\max_{a \in A} q(s,a) = arg\max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V(s') \right)$$

将这两个函数合并可以得到这样一个公式（也就是将策略迭代中的策略评估和策略提升两个步骤合二为一）

$$V(s) \leftarrow \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V(s') \right)$$

这个公式实际上就是贝尔曼最优方程

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

满足贝尔曼最优方程的策略一定是最优策略，这也不难解释为什么可以通过价值迭代找到最优策略。

价值迭代算法的伪代码如下

算法参数：小阈值 $\theta > 0$ ，用于确定估计量的精度

对于任意 $s \in S^+$ ，任意初始化$V(s)$，其中 $V(\text{终止状态}) = 0$

循环：

$\Delta \leftarrow 0$

对每一个 $s \in S$ 循环：

$v \leftarrow V(s)$

$$V(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a V(s') \right)$$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

直到 $\Delta < \theta$

输出一个确定的策略 $\pi \approx \pi_*$ ，使得

$$\pi(s) = \operatorname{argmax}_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \right)$$

(2)算法实现

价值迭代本质上是策略迭代的策略评估只进行一次退化得到，经过与贪婪算法的整合之后就得到了贝尔曼最优方程，因此价值迭代的本质就是通过使用 Bellman 方程重复计算每个状态的期望值并找到所有可能动作的最大期望值来更新 v 列表，直到新旧值函数之间的差值小于 theta 阈值，收敛后得到的策略就是最优策略。

下面的价值迭代通过迭代计算值函数和最优策略求解最优策略

- 将所有状态对应的价值函数和阈值 theta 初始化为 0，while 循环退出的条件是值函数的最大差值小于阈值 theta；
- new_v 列表将存储迭代过程中的更新的值函数；
- 对于每个状态 s，算法使用当前值函数来计算每个动作 a 的期望值 Q，这通过迭代在状态 s 中采取 action 的所有可能结果完成，采取的 action 由状态转移矩阵确定；
- 一旦已经为状态 s 中的所有动作计算了 Q 值，该算法就将状态 s 的值设置为所有动作的最大 Q 值，这对应于贝尔曼最优方程；

```
        while 1:
            delta = 0
            new_v = [0] * self.env.ncol * self.env.nrow
            # 对于每个状态s，函数计算每个动作a的期望值，并选择使该值最大化的动作
            for s in range(self.env.ncol * self.env.nrow):
                exp_list = []# 用于存储每个状态-动作对的Q值，Q值是通过在特定状态s中采取特定动作a而获得的未来奖励的预期总和
                for a in range(4):# 对于每个动作a
                    exp = 0
                    for res in self.env.P[s][a]:# 使用当前值函数计算采取该操作的预期值
                        exp += res[0] * (res[2] + self.gamma * self.V[res[1]]*(1-res[3]))
                    exp_list.append(exp)# 将此值添加到列表qsa_list中
                    # 一旦已经为状态s中的所有动作计算了Q值，该算法就将状态s的值设置为所有动作的最大Q值，这对应于贝尔曼最优方程
                new_v[s] = max(exp_list)
                # 将new_v中的状态值设置为qsa_list中的最大值 ，该方程指出，一个状态的最优值等于所有可能行动的未来回报的最大预期总和
                delta = max(delta, abs(new_v[s] - self.V[s]))
            self.V = copy.deepcopy(new_v)
            if delta < self.theta:
                break
        print('Value Iteration Done!')
```

计算出值函数之后，算法通过选择每个状态 s 下使得期望最大对应的 action 导出策略 policy，如果有多个动作具有相同的期望值，则算法为策略中的这些动作分配相等的概率；

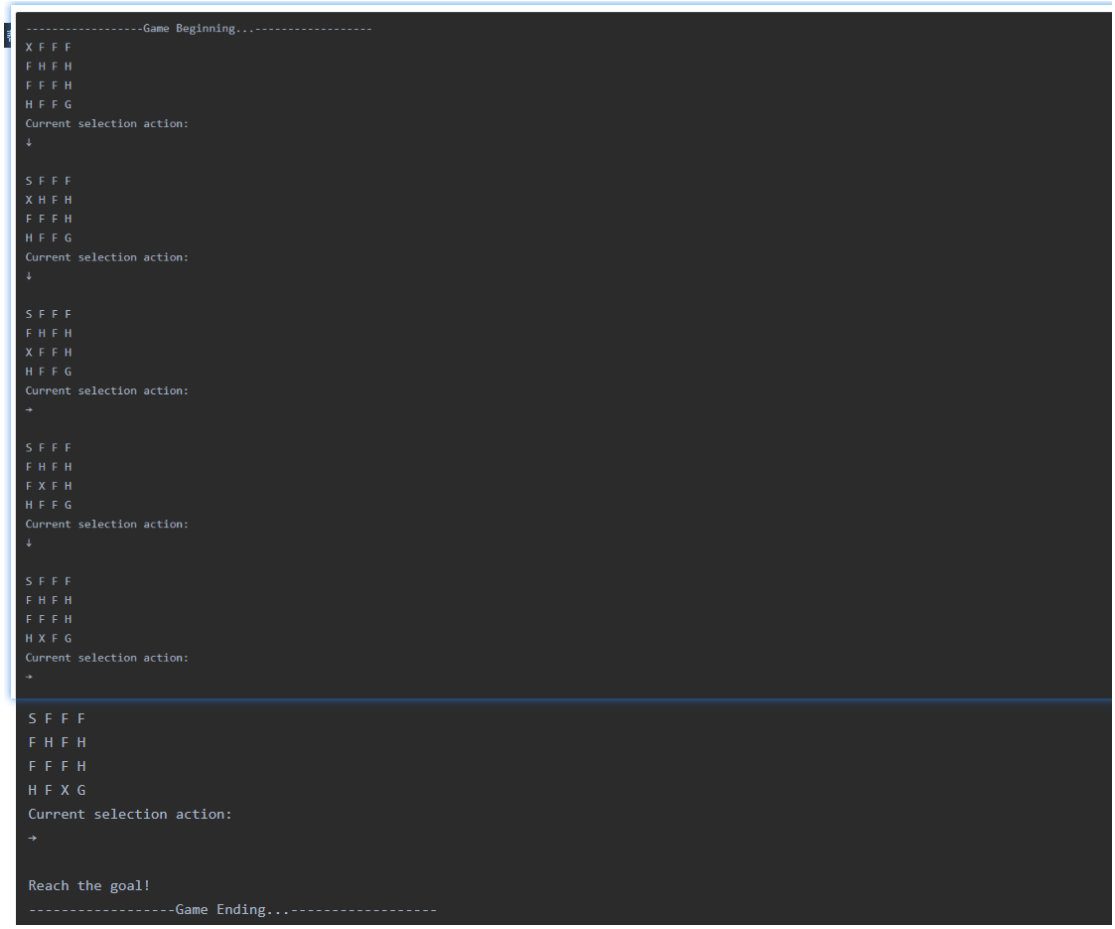
```
        for s in range(self.env.ncol * self.env.nrow):
            exp_list = []
            for a in range(4):
                exp = 0
                for res in self.env.P[s][a]:
                    # 使用最优值函数self.v计算每个状态-动作对的Q值
                    exp += res[0] * (res[2] + self.gamma * self.V[res[1]]*(1-res[3]))
                exp_list.append(exp)
            max_exp = max(exp_list)
            cnt_max = exp_list.count(max_exp)
            # 将具有最大Q值的动作的概率设置为1，并将所有其他动作的概率设为0来提取每个状态的最优策略
            # 让相同的动作价值均分概率
            self.policy[s] = [1 / cnt_max if q == max_exp else 0 for q in exp_list]
        print('Policy Improvement Done!')
```

(3)算法实践

此处的 train 和 test 方法与前面策略迭代相同，只需要将 train 的参数从 PolicyIteration 修改为 ValueIteration，train 的结果如下

```
ValueIteration Beginning...
Value Iteration Done!
Policy Improvement Done!
------------------OptimalValue------------------
 0.590  0.656  0.729  0.656
 0.656  0.000  0.810  0.000
 0.729  0.810  0.900  0.000
 0.000  0.900  1.000  0.000
------------------OptimalPolicy------------------
State 0 -> Optimal Action:↓
State 0 -> Optimal Action:→
State 1 -> Optimal Action:→
State 2 -> Optimal Action:↓
State 3 -> Optimal Action:←
State 4 -> Optimal Action:↓
State 5 -> Optimal Action: Hole
State 6 -> Optimal Action:↓
State 7 -> Optimal Action: Hole
State 8 -> Optimal Action:→
State 9 -> Optimal Action:↓
State 9 -> Optimal Action:→
State 10 -> Optimal Action:↓
State 11 -> Optimal Action: Hole
State 12 -> Optimal Action: Hole
State 13 -> Optimal Action:→
State 14 -> Optimal Action:→
State 15 -> Optimal Action: End
```

可以看到价值迭代的结果和策略迭代的结果完全一致，从理论上来说也应当是一致的，因此验证了算法的正确性，根据 train 得到的策略进行 test 的结果如下，这与策略迭代的移动轨迹也是相同的



1.3 Q-learning 算法

(1)算法介绍

Q 学习考虑基于动作价值函数 Q(s,a)的离轨学习，其用于自举的动作选择的是目标策略的后续状态对应的动作，而当前状态选择的动作是行为策略选择的当前状态对应的动作，因此 Q 学习的策略迭代表达式如下

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)\right)$$

行为策略的动作　　目标策略的动作　　行为策略的动作

尽管观察者和游戏玩家共享 Q 函数，但它们对策略的优化是不同的，目标策略的优化使用的是最朴素的贪婪优化方式（无探索行为）

$$\pi(S_{t+1}) = \underset{a'}{\operatorname{argmax}}\, Q(S_{t+1}, a')$$

而行为策略的优化使用的是 ε-greedy 的方式（有探索行为，目标策略和行为策略不同的优化方式实现了 Q 学习在遵循探索性策略的同时学习最优策略）

将贪婪策略代入 Q 学习的策略迭代表达式，可以得到如下式子

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

可以看到该表达式的形式非常类似贝尔曼最优方程；事实上，Sarsa 中使用的是贝尔曼期望方程进行更新，而 Q 学习使用的是贝尔曼最优方程进行更新，因此 Q 学习收敛时一定会收敛到最优值。

Q 学习算法的伪代码如下

1: 随机初始化$Q(s,a), \forall s \in S, a \in A(s)$,将$Q(terminal-state, \cdot) = 0$

2: REPEAT （对于每一个episode）：

3: 初始化 S

4: REPEAT （对于episode中的每一步）：

5: 根据从Q得到的策略(e.g., $\epsilon - greedy$)在状态S时选择动作A

6: 执行动作A, 观察到 R, S'

7: $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',A') - Q(S,A)\right]$

8: $S \leftarrow S'$

9: 直到S为终止状态

(2)算法实现

Q-learning 算法的核心在于在每个 episode 上迭代并根据当前 state 和 Q 表选择 action，同时根据收到的 rewards 和下一状态的最大 Q 值更新 Q 表，最终理论上会得到一张对应最优策略的 Q 表。

- 初始化 Q 表（存储每个<状态，动作>对的 Q 值）并在指定数量的 episodes 上迭代，每个 episode 都是 agent 和 env 之间的完整交互；
- 在每个 episode 中 agent 使用 ε 贪婪探索策略基于 Q 表中的 Q 值来选择动作， ε 值随着时间的推移衰减；
- Q 值使用贝尔曼方程更新，基于获得的 rewards 和下一个状态的最大 Q 值来更新；

```
def Q_Learning(self):
    for episode in range(self.episodes):
        total_reward = 0# 初始化当前episode的reward为0
        state = self.env.reset()# 重置当前状态为起始状态
        for step in range(self.max_steps):# 在max_steps_per_set范围内的每个步骤上循环
            rand = random.uniform(0,1)
            # 如果随机生成的数字小于探索率，则使用环境的get_random_action () 方法选择一个随机操作
            if rand < self.epsilon:
                action = self.env.get_random_action().value
            # 否则使用numpy的argmax () 方法为当前状态选择Q值最高的动作，以检索Q值最高动作的索引
            else:
                action = np.argmax(self.Q[state,:])
            # 使用环境的step方法将所选action作用在环境上，step返回下一个状态、采取动作的奖励以及指示事件是否完成的布尔值
            new_state,reward,done = self.env.step(action)
            # 更新当前<状态，动作>对的Q值
            self.Q[state][action] = self.Q[state][action] * (1 - self.learning_rate) + \
                                    self.learning_rate * (
                                            reward + self.discount_rate * np.max(
                                        self.Q[new_state, :]))
            # 更新当前状态
            state = new_state
            # 用从action中获得rewards更新当前episode的总奖励
            total_reward += reward
            if done:# 假如该episode已完成，则循环中断
                break
        # 每次episode结束后，通过使用指数衰减函数将探索率从最大值逐渐降低到最小值来更新探索率
        self.epsilon = self.epsilon_min + (self.epsilon_max - self.epsilon_min) * np.exp(-self.epsilon_decay * episode)
        # 将当前eposide获得的总奖励存储在一个名为total_reward的列表中
        self.rewards.append(total_reward)
```

(3)算法实践

该部分同样分为两部分，train 部分初始化环境后，传入 env 以实例化一个 QlearningAgent 类对象，将其命名为 agent 也就是小机器人，调用小机器人的 Q_Learning 方法即可开始执行 Q-Learning 算法得到最优策略，通过 agent 的 print_result 方法可视化训练结果。

```
def train():
    env = FrozenLake()
    agent = QlearningAgent(env)
    agent.Q_Learning()
    agent.print_results(['←', '↓', '→', '↑'],[5, 7, 11, 12], [15])
    return agent
def test(agent):
    print('------------------Game Beginning...-------------------')
    agent.latest_iteration()
    print('------------------Game Ending...-------------------')
```

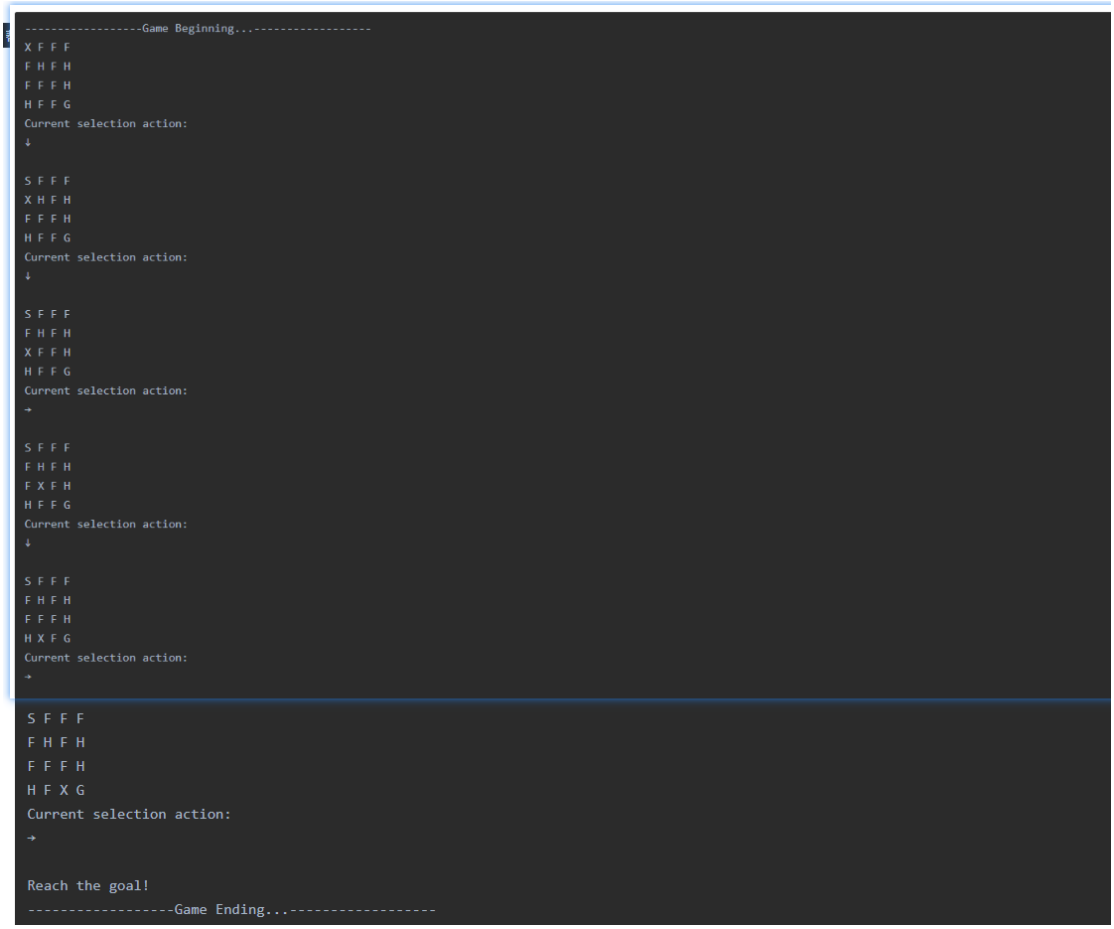
test 部分调用 agent 的 latest_iteration 方法对训练得到的最优策略（实际上是基于最后一次迭代得到的 Q 表，但当时已经收敛因此可以看作是最优策略）进行测试，latest_iteration 部分的基本思想就是 agent 从起点开始根据训练得到的 Q 表，根据当前状态选择 action 中概率最大的 action 执行，以此类推，最终到达终点

```
def latest_iteration(self):
     state= self.env.reset()
     for step in range(self.max_steps):
        self.env.print_current_state()
        time.sleep(1)
        action = np.argmax(self.Q[state,:])
        print('当前选择动作: ')
        self.print_action(action)
        print('\n')
        new_state,reward,done = self.env.step(action)
        state = new_state
        if done:
            print('Reached the goal')
            break
```

先看 train 得到的结果，分别对 Q 表、最优策略以及平均奖励做了可视化。
其中最优策略就是选择 Q 表中的每个状态的 action 列表中概率最大的 action_index 对应的 action，通过平均奖励也可以看出在训练了 8000 个 episodes 之后 agent 就已经确定了最终的最优策略，故奖励不再增大，代表整个算法收敛。

```
------------------Q-Table------------------
[[9.41480149e-01 9.50990050e-01 9.32065348e-01 9.41480149e-01]
 [9.41480149e-01 0.00000000e+00 7.29121367e-01 9.13888936e-01]
 [8.76578874e-01 2.08886992e-01 1.28175264e-02 8.17249166e-02]
 [1.60521527e-01 0.00000000e+00 6.43730402e-07 6.43730402e-07]
 [9.50990050e-01 9.60596010e-01 0.00000000e+00 9.41480149e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 8.83844680e-01 0.00000000e+00 9.06726299e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [9.60596009e-01 0.00000000e+00 9.70299000e-01 9.50990049e-01]
 [9.60596009e-01 9.80100000e-01 9.80099987e-01 0.00000000e+00]
 [9.28975931e-01 9.90000000e-01 0.00000000e+00 6.85405926e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 9.80099955e-01 9.90000000e-01 9.70298900e-01]
 [9.80099980e-01 9.89999995e-01 1.00000000e+00 9.80099955e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
------------------OptimalPolicy-----------------
State 0 -> Optimal Action:↓
State 1 -> Optimal Action:←
State 2 -> Optimal Action:←
State 3 -> Optimal Action:←
State 4 -> Optimal Action:↓
State 5 -> Optimal Action: Hole
State 6 -> Optimal Action:↓
State 7 -> Optimal Action: Hole
State 8 -> Optimal Action:→
State 9 -> Optimal Action:↓
State 10 -> Optimal Action:↓
State 11 -> Optimal Action: Hole
State 12 -> Optimal Action: Hole
State 13 -> Optimal Action:→
State 14 -> Optimal Action:→
State 15 -> Optimal Action: End
------------------AverageRewards-----------------
1000 :  0.265
2000 :  0.741
3000 :  0.9
4000 :  0.96
5000 :  0.973
6000 :  0.988
7000 :  0.988
8000 :  0.991
9000 :  0.991
10000 :  0.991
```

使用收敛之后得到的 Q 表对 agent 的行为进行指导，我们得到在 test 上的行进路线（这个路线和策略迭代、价值迭代是相同的）



2.实验分析

2.1 概念区分

(1)价值迭代&策略迭代

价值迭代和策略迭代时动态规划中用于求解最优策略的两种迭代算法，这两个算法都属于动态规划算法，同时这两个方法解决的都是一个已知的 MDP(表示已知表示状态之间的转移概率和在某个状态下得分的期望都已知)问题。

关于价值迭代和策略迭代的介绍在前面的算法介绍中已经详细介绍过，两者之间的主要区别在于，价值迭代通过迭代改进价值函数的初始估计来计算最优值，而策略迭代通过迭代改善初始策略来计算最优策略。

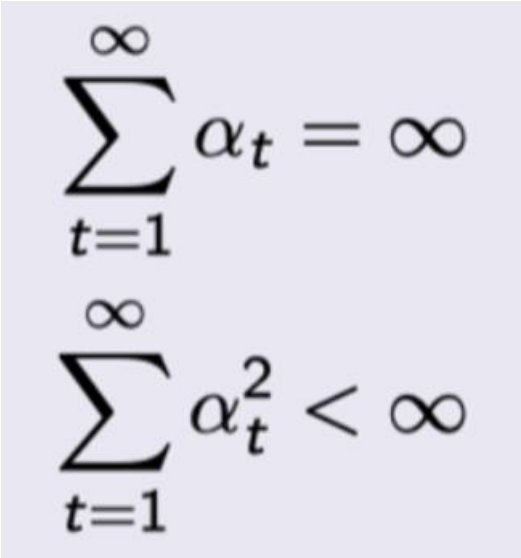
另一方面，因为价值迭代并不是在已知策略上进行改进，而是直接基于价值寻找一个最优策略，因此迭代使用贝尔曼最优方程进行回溯得到的最优策略 Π，可能完全是一个全新的策略，而策略迭代中的策略是已知的策略的改进。

(2)Sarsa&Q-learning

Sarsa 算法和 Q 学习算法都是无模型控制算法，区别在于 Sarsa 是在轨 TD 控制算法，Q 学习是离轨 TD 控制算法。在此之前需要先区分在轨和离轨的区别，在轨学习可以理解为自己作为玩家和观察者，边打游戏边升级策略；离轨学习可以理解为自己作为观察者观察玩家打游戏，升级自己的策略。

Sarsa 算法不一定收敛到全局最优（可能收敛到局部最优），Sarsa 收敛于最优价值函数当且仅当满足：

1. 任何时候的策略都满足 GLIE 特性
2. 步长系数 α ~t~满足



$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

而 Q 学习收敛一定是收敛到最优值，原因是 Sarsa 中使用的是贝尔曼期望方程进行更新，而 Q 学习使用的是贝尔曼最优方程进行更新。

在 Q 值更新的方面，Sarsa 的 Q 值更新公式中，R+ γ Q(S',A')可以视为目标 Q 值，Sarsa 每次都是在下一次采取实际动作之后再更新 Q 值，A'表示下一次采取的实际 action，Sarsa 的每次更新都是在动作实际发生以后。

因为 Sarsa 是在实际动作发生后更新 Q 值，而采取 ε -greedy 策略会有一定概率进行探索，因此表现在悬崖问题中，Sarsa 会选择离悬崖最远的路径（这样是最安全的）。

Q 学习的 Q 值更新公式表示，Q 学习在更新 Q 值的时候下一步动作是不确定的，它会选择 Q 值最大的动作进行下一步的更新，表现在悬崖问题中，Q 学习会选择离悬崖最近的路径，但是因为掉落次数较多所以分数低于 Sarsa。

总的来说，Sarsa 表现得更为胆小，因为它会记住每一次错误的探索，它会对错误较为敏感，而 Q-learning 只在乎 Q 值的最大化，因此 Q-learning 会十分贪婪，表现得十分大胆。

2.2 随机动作

Frozen Lake 游戏有一个很有意思的地方，在于可以将冰面的属性设置为 slippered，这意味着 agent 选择执行的动作并不一定会执行，而有可能以等概率的形式执行除了选择方向反方向的三个方向之一，可以通过下面的代码验证

```
import gymnasium as gym
import copy
import numpy as np
from copy import deepcopy

env = gym.make("FrozenLake-v1",is_slippery=False)  # 创建环境
for a in env.P[14]:  # 查看终点左边一格的状态转移信息 -- 每个动作都会等概率滑行到3种可能结果（一行对应一个动作的3种可能）
    print(env.P[14][a])
```

输出结果如下

```
[(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)]
```

上面代码的意思是指查看 14 状态的状态转移信息，这意味着每个动作都会等概率({'prob': 0.3333333333333333}))滑行到 3 种可能结果（一行对应一个动作的 3 种可能）。

那么当冰面变得光滑的时候动态规划算法或 Q-learning 算法是否还起作用呢？我们选择策略迭代算法进行检验，只需要将 env 的初始化中 is_slippered 修改为 True 即可。

执行完策略迭代的 train 算法后得到如下结果

```
------------------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
------------------最优策略------------------
State 0 -> Optimal Action:←
State 1 -> Optimal Action:↑
State 2 -> Optimal Action:←
State 3 -> Optimal Action:↑
State 4 -> Optimal Action:←
State 5 -> Optimal Action: Hole
State 6 -> Optimal Action:←
State 6 -> Optimal Action:→
State 7 -> Optimal Action: Hole
State 8 -> Optimal Action:↑
State 9 -> Optimal Action:↓
State 10 -> Optimal Action:←
State 11 -> Optimal Action: Hole
State 12 -> Optimal Action: Hole
State 13 -> Optimal Action:→
State 14 -> Optimal Action:↓
State 15 -> Optimal Action: End
```

这个结果比较令人惊讶，如处于状态 0 的时候选择的是向左移动，直观感受是 agent 会移动出 grid，但是因为冰面光滑的原因，导致 agent 要么向左要么向上要么向下，因为向左和向上都会出 grid，但是向下会得到更高的价值。根据冰面滑行的原则，是可以理解的为什么不选择直观方向的。
在冰面光滑的条件下 agent 的移动就不是 action 唯一指定了，故 test 的结果是非常长的一串不断尝试的行为，因为动作具有随机性所以 agent 可能移出 grid、掉进冰窟，这就导致 agent 需要重新从起点开始，因此 agent 明显尝试的次数相较于确定性动作多得多。

3.实验总结

本次实验主要涉及三个任务，分别是基于马尔可夫过程价值函数的计算、基于动态规划的强化学习算法的实现以及 Q 学习算法的实现，这三个任务从最简单的有限马尔可夫决策过程到动态规划再到无模型控制，包含了强化学习课程大部分的知识点，通过对这三个任务的处理也让我对强化学习的知识点的掌握更进一步。
在计算价值函数的过程中遇到的一个问题是使用同步更新方式还是异步更新方式，最终我选择了同步更新方式（因为衰减因子的存在异步更新会面临非常多小数点的处理）。
在实现强化学习算法的过程中，首先遇到的困难就是关于 Frozen Lake 环境的理解，首先需要配置 gymnasium 库以及相关依赖，之后结合官网的 document 深入理解了实验环境后结合算法伪代码分别实现了 PolicyIterationAgent 类、ValueIterationAgent 类和 QlearningAgent 类，解决了 agent 在 Frozen Lake 上的路线问题。