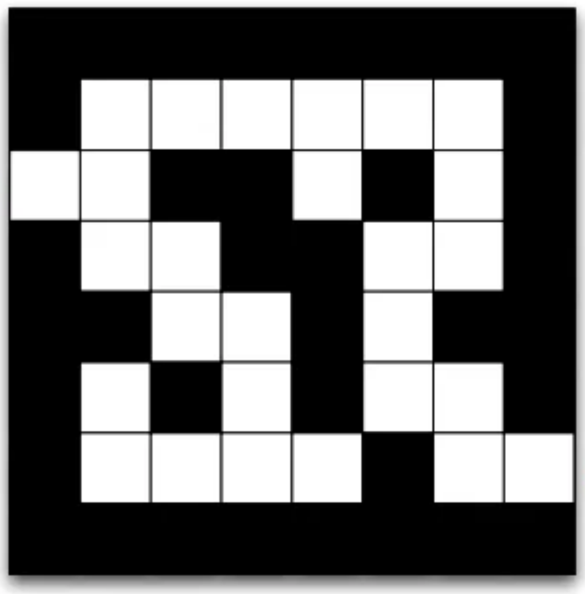
Assignment 1

21307035 邓栩瀛

1、实验内容

Solve the Maze Problem using Policy Iteration or Value Iteration.



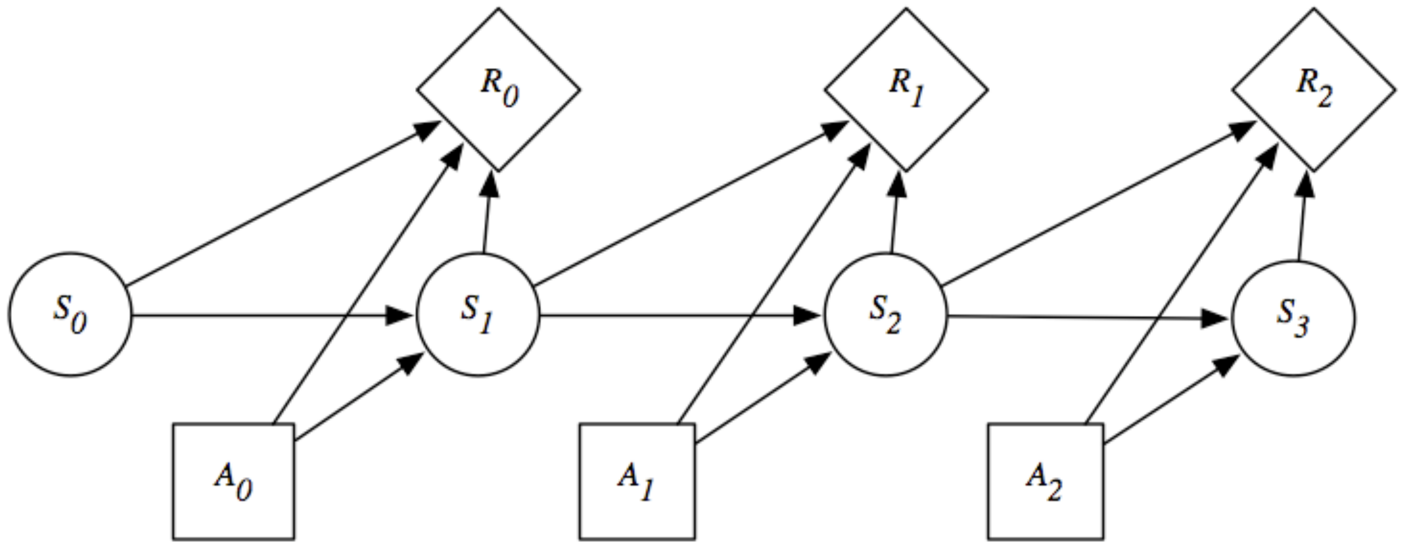
2、MDP建模

一个马尔可夫决策过程（Markov Decision Process, MDP）由一个四元组(S, A, P_{sa}, R)构成：

- S: 表示状态集（states）
- A:表示一组动作（actions）
- P_{sa}: 表示状态转移概率。表示的是在当前$s \in S$状态下，经过$a \in A$作用后，会转移到的其他状态的概率分布情况。比如，在状态s下执行动作a，转移到s'的概率可以表示为$p(s'|s, a)$
- R: $S \times A \to \mathbb{R}$，$R$是回报函数（reward function），回报函数有时也写作状态S的函数（只与S有关），这样的话，R可以简化为$R: S \to \mathbb{R}$。

MDP 的动态过程如下：某个智能体(agent)的初始状态为S_0，然后从 A 中挑选一个动作a_0执行，执行后，agent 按P_{sa}概率随机转移到了下一个s_1状态，$s_1 \in P_{s_0a_0}$。然后再执行一个动作a_1，就转移到了s_2，接下来再执行$a_2 \ldots$

如果回报r是根据状态s和动作a得到的，则MDP还可以表示成下图：



设一个状态的价值为$v(s)$，则$v(s)$与当前状态的奖励$R(s)$和此状态即将转移状态的$v(s')$有关，设s转移到s'的概率为$P(s'|s)$，则bellman equation为：

$$v(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)V(s') \tag{1}$$

加上action的过程即为MDP，即多了一个动作价值函数，价值函数v (s) v(s)v(s)与动作价值函数的关系为：

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) q^{\pi}(s,a) \tag{2}$$

$v(s), q(s,a)$对应的贝尔曼方程为：

$$\begin{aligned} v(s) &= \sum_{a \in A} \pi(a|s)(R(s) + \gamma \sum_{s' \in S} P(s'|s)V^{\pi}(s')) \\ q^{\pi}(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in A} \pi(a'|s') q^{\pi}(s',a') \end{aligned} \tag{3}$$

MDP寻找的是在任意初始条件s下，能够最大化值函数的策略π。最优策略表示为：

$$\pi = argmax_{\pi} V^{\pi}(s), (\forall s) \tag{4}$$

与最优策略π对应的状态值函数V与动作值函数Q之间存在如下关系：

$$V = max_a Q(s,a) \tag{5}$$

而求解最优策略，则可以用策略迭代（policy iteration）或值迭代（value iteration）的方式，我在本次实验中采用的是值迭代的方式。

在本实验中

（1）状态集（S）

- 迷宫的每一个可行走的格子都表示一个状态。在8x8的迷宫中，除去障碍格子（值为1）外，其余的格子都是0状态。
- 终止状态为迷宫中的终点，即坐标为 (6, 8) 的格子。

（2）动作集（A）

- 共有四种动作，分别是向上、向右、向下和向左移动，表示为：$A = \{上, 右, 下, 左\}$
- 每个动作会引导智能体移动到相邻的格子，除非遇到迷宫边界或障碍。

（3）转移概率矩阵（$P(s'|s,a)$）

- 在给定状态 s 和动作 a 的情况下，转移到下一个状态 $s\prime$ 的概率。若下一个状态 $s\prime$不存在障碍，概率为1；若无法移动，则该方向的概率为0。
- 由于终点状态不再转移，特殊处理了该状态的转移概率。

（4）奖励函数（R）

每个状态的奖励为 -1，除非到达终止状态，奖励为20000。这种设定保证了路径越短，累积的负奖励越少，智能体会尽量找到最快路径。

3、算法说明

本实验采用值迭代Value Iteration算法来解决MDP问题。值迭代通过迭代更新每个状态的值函数，最终收敛于最优值函数。

1. 值迭代算法流程

   1.初始化状态值函数 V(s) 为 0。

   2.对于每个状态 s，在每次迭代中执行以下操作：
   •计算每个可能动作 a 导致的期望值：

$$Q(s,a) = \sum_{s\prime} P(s\prime|s,a)[R(s,a,s\prime) + \gamma V(s\prime)] \tag{6}$$

其中，γ 是折扣因子（设定为0.9），表示未来奖励的权重。
•对于状态 s，选择使得 Q(s, a) 最大的动作，并更新值函数 V(s)。

3.通过反复迭代，最终状态值函数 V(s) 收敛。

2. 策略提取

在值函数收敛后，使用贪婪策略提取最优策略，对于每个状态 s，选择最优动作：

$$\pi(s) = \arg\max_a Q(s,a) \tag{7}$$

该策略指引智能体在每个状态下执行使得未来期望收益最大的动作。

值迭代的算法伪代码如下：

input : reward function $r(s)$, transitional model $p(s'|s,a)$, discounted factor γ, convergence threshold θ

output: optimal policy π^*

initialize $v(s)$ with zeros
converge $\leftarrow$ false
while converge $=$ false **do**
    $\Delta \leftarrow 0$
    for $s \in S$ **do**
        temp $\leftarrow v(s)$
        $v(s) \leftarrow r(s) + \gamma \max_a \sum_{s'} p(s'|s,a)v(s')$
        $\Delta \leftarrow \max(\Delta, |\text{temp} - v(s)|)$
    end
    if $\Delta < \theta$ **then**
        converge $\leftarrow$ true
    end
end
for $s \in S$ **do**
    $\pi^*(s) \leftarrow \text{argmax}_a \sum_{s'} p(s'|s,a)v(s')$
end
return π^*

4、关键代码

1.初始化迷宫和参数

```
dim = 8
num_states = []
terminal_state = (6, 8)

maze = np.array([
    [1., 1., 1., 1., 1., 1., 1., 1.],
    [1., 0., 0., 0., 0., 0., 0., 1.],
    [0., 0., 1., 1., 0., 1., 0., 1.],
    [1., 0., 0., 1., 1., 0., 0., 1.],
    [1., 1., 0., 0., 1., 0., 1., 1.],
    [1., 0., 1., 0., 1., 0., 0., 1.],
    [1., 0., 0., 0., 0., 1., 0., 0.],
    [1., 1., 1., 1., 1., 1., 1., 1.]
])
```

- `dim` 表示迷宫的维度（8x8）。
- `num_states` 用于保存所有非障碍的状态。
- `terminal_state` 是迷宫的终点 (6, 8)。
- `maze` 是一个二维数组，表示迷宫。1 表示障碍，0 表示可走的空白格子。

2.定义动作

```
actions = [(-1, 0), (0, 1), (1, 0), (0, -1)]  # 上、右、下、左
directions = ["↑", "→", "↓", "←"]  # 用于策略可视化
P_a = np.zeros((dim * dim, 4))  # 动作概率矩阵
state_map = {}  # 状态映射表
```

- `actions` 定义了四个方向的动作（上下左右）。
- `directions` 是动作方向的符号，用于输出策略时表示方向。
- `P_a` 是一个维度为 (64, 4) 的数组，表示每个状态在每个动作下的转移概率。
- `state_map` 是一个字典，用于将二维坐标映射到状态编号。

3.初始化函数

```
def init():
    for i in range(dim):
        for j in range(dim):
            if maze[i][j] == 0:
                num_states.append((i, j))
                state_map[(i, j)] = len(num_states) - 1
```

`init()` 函数遍历迷宫，将所有可行的状态（即 0 的位置）保存到 `num_states`，并在 `state_map` 中记录对应状态编号。

```
    for i in range(dim * dim):
        x, y = divmod(i, dim)
        if maze[x][y] == 1:
            continue
```

遍历每个位置，如果是障碍则跳过。

```
        sum_prob = 0
        for k, (dx, dy) in enumerate(actions):
            nx, ny = x + dx, y + dy
            if 0 <= nx < dim and 0 <= ny < dim and maze[nx][ny] == 0:
                P_a[i][k] = 1.
                sum_prob += 1.
```

对于每个位置，尝试在四个方向上移动，检查是否是可行状态。若可行，则在 `P_a` 中标记对应的动作概率。

```
        if sum_prob > 0:
            P_a[i] /= sum_prob
```

正常化概率，使每个动作的概率之和为 1。

```
    P_a[55][1] = 0.5
    P_a[55][3] = 0.5
```

针对终点状态手动处理，使其左右动作的概率均为 0.5。

4.值迭代算法

```
def value_iteration(gamma=0.9, num_of_iterations=10000):
    N_STATES = len(num_states)
    N_ACTIONS = len(actions)
    error = 0.0001

    values = np.zeros(N_STATES)  # 初始化值函数
    rewards = [-1] * N_STATES  # 每个状态的奖励为 -1
```

- `gamma` 为折扣因子，表示未来奖励的权重。
- `num_of_iterations` 为迭代次数。
- `values` 存储每个状态的值函数。
- `rewards` 为每个状态的即时奖励，设为 -1 表示惩罚（鼓励更快到达终点）。

```
    for _ in range(num_of_iterations):
        values_tmp = values.copy()

        for idx in range(N_STATES):
            v_a = []
            x, y = num_states[idx]
            s = x * dim + y
```

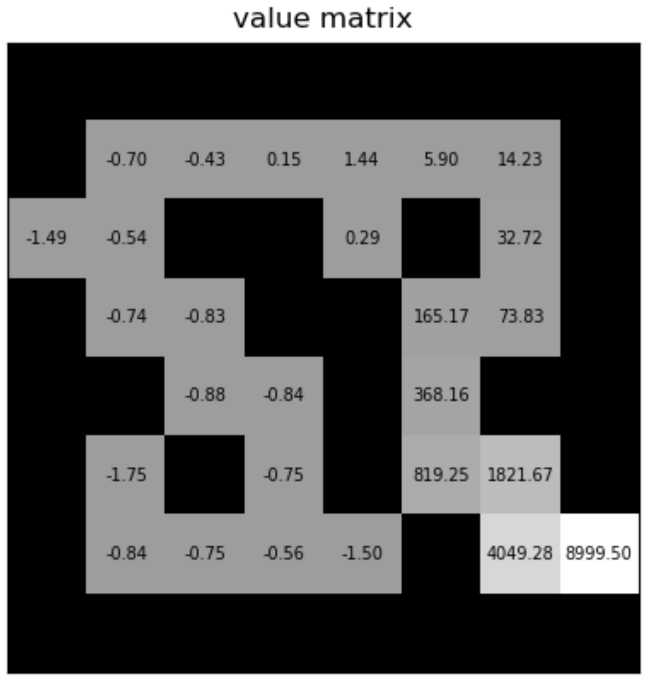
通过多次迭代，更新每个状态的值函数。

```
            for a in range(N_ACTIONS):
                if P_a[s][a] != 0:
                    nx, ny = x + actions[a][0], y + actions[a][1]
                    if (nx, ny) == terminal_state:
                        v_a.append(P_a[s][a] * (rewards[idx] + gamma * 20000))
                    else:
                        s1 = state_map[(nx, ny)]
                        v_a.append(P_a[s][a] * (rewards[idx] + gamma * values_tmp[s1]))
            values[idx] = max(v_a)
```

- 对于每个状态，计算所有动作带来的可能值，并选择最大的值更新值函数。
- 如果到达终点，设置特殊奖励 20000。

5、实验结果

Value Matrix



Policy Matrix