中国科学技术大学
University of Science and Technology of China

《人工智能数学原理与算法》
第 7 章：强化学习

7.3 动态规划算法

吉建民
jianmin@ustc.edu.cn

目录

01 动态规划算法：介绍

02 动态规划算法：策略评估

03 动态规划算法：策略迭代

04 动态规划算法：价值迭代

目录

01 动态规划算法：介绍

02 动态规划算法：策略评估

03 动态规划算法：策略迭代

04 动态规划算法：价值迭代

什么是动态规划算法?

- **动态规划算法 (Dynamic Programming)**：是一种求解多阶段决策过程最优化问题的方法。在动态规划中，通过把原问题分解为相对简单的子问题，先求解子问题，再由子问题的解而得到原问题的解
  - 动态规划算法的要求：具有最优子结构和子问题重叠
  - **最优子结构**（Optimal substructure）：适用最优原理，最优解可分解为子问题的解
  - **重叠子问题**（Overlapping subproblems）：子问题会多次出现，其解可缓存复用

马尔可夫决策过程满足上述特性：贝尔曼方程提供了递归分解方式，价值函数存储并复用求解结果。

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) \left(R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V_\pi(s') \right), \quad Q_\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) \sum_{a' \in A} \pi(a' \mid s') Q_\pi(s',a'),$$

$$V^*(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^*(s') \right), \quad Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) \max_{a' \in A} Q^*(s',a').$$

基于动态规划的“规划”

- 在已知完整 MDP 模型下，动态规划算法解决如下“规划”问题
  - **预测（Prediction）**
    - 输入：MDP 模型 $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ 和给定策略 π
    - 输出：该策略的价值函数 $v_\pi(s)$
  - **控制（Control）**
    - 输入： MDP 模型 $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
    - 输出：最优价值函数 v_* 或最优策略 π_*

动态规划算法的应用

- **动态规划可以解决许多问题，例如：**

  - 调度优化：如作业车间调度（Job-Shop Scheduling），将每道工序拆分为“机器 × 时段”子状态，计算最小完工时间或最小延误成本

  - 图算法：如最短路径，图中所有节点对的最短路径

  - 资源分配 / 背包类：如 0-1 背包问题，在给定容量或预算约束下，寻找物品组合，使总体收益最大或成本最小

本章重点讨论如何使用动态规划来求解 MDP，**具体涉及策略评估、策略迭代和值迭代。**

目录

01 动态规划算法：介绍

02 动态规划算法：策略评估

03 动态规划算法：策略迭代

04 动态规划算法：价值迭代

迭代策略评估（Iterative Policy Evaluation）

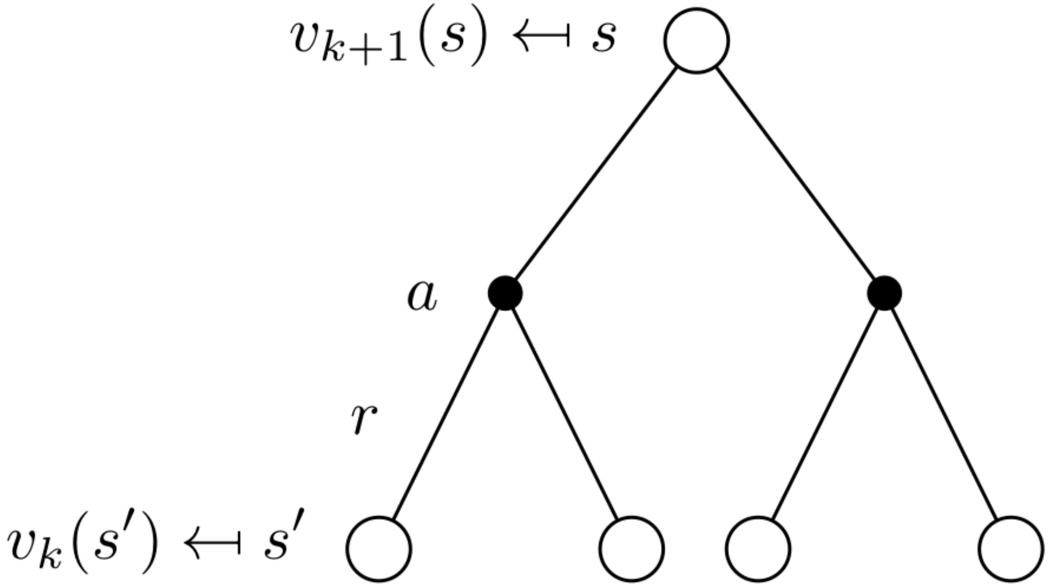
- 预测问题：给定 MDP 模型，评估策略 π，即，计算其价值函数 v_π
- 迭代策略评估：反复应用贝尔曼期望方程进行迭代

$$v_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a \mid s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

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

- 同步（synchronous）迭代：$k+1$ 步的价值函数 $v_{k+1}(s)$ 由 k 步的价值函数 $v_k(s)$ 针对所有状态 $s \in \mathcal{S}$ 统一进行更新
- 扩展阅读：收缩映射定理（Contraction Mapping Theorem）保证上述迭代过程一定会收敛到唯一的不动点 v_π

迭代策略评估



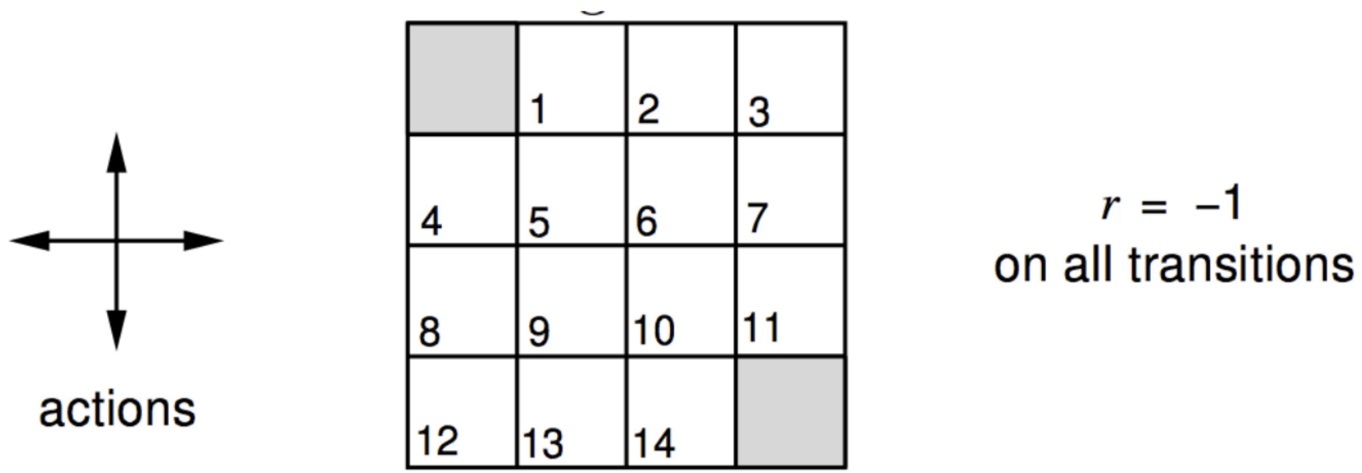
$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}^{k+1} = \boldsymbol{\mathcal{R}}^{\boldsymbol{\pi}} + \gamma \boldsymbol{\mathcal{P}}^{\boldsymbol{\pi}} \mathbf{v}^k$$

迭代策略评估算法

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

示例：网格世界



- 在网格世界中评估一个随机策略
- 网格世界 MDP：
  - 无折扣，γ=1
  - 1 到 14 为非终止状态，两块阴影为终止状态
  - 到达终止状态之前，每行动一步，奖励值为 -1
  - 四个行动分别为向北、南、东、西，智能体根据行动在格子间移动，执行失败则保持在原格子

智能体遵循均匀随机策略，即向各个方向（北、南、东、西）移动的概率均为 0.25， 也就是

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

网格世界：迭代策略评估

v_k for the Random Policy

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

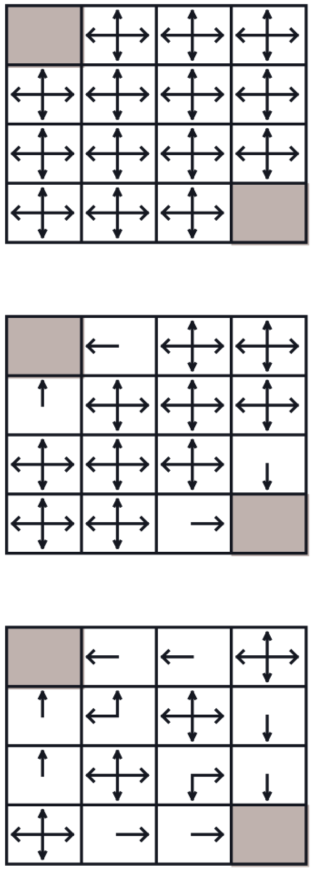
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

Greedy Policy w.r.t. v_k



random policy

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a) \\ 0 & \textit{otherwise} \end{cases}$$

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

网格世界：迭代策略评估

$k = 3$

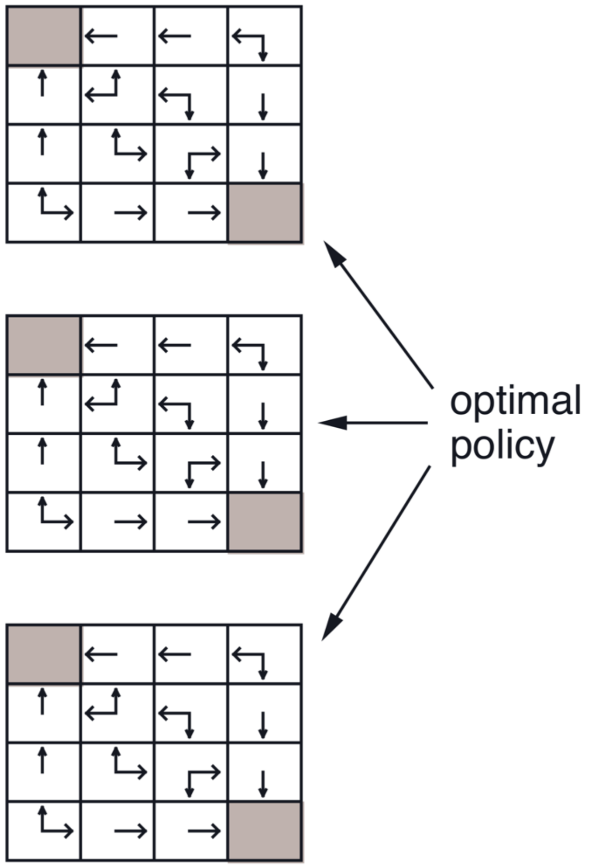
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



目录

01 动态规划算法：介绍

02 动态规划算法：策略评估

03 动态规划算法：策略迭代

04 动态规划算法：价值迭代

如何改进策略?

- **步骤 1：策略评估**
  - 给定当前策略 π
  - 计算其价值函数 v_π：例如，迭代策略评估，线性代数直接求解等

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + ...|S_t = s\right]$$

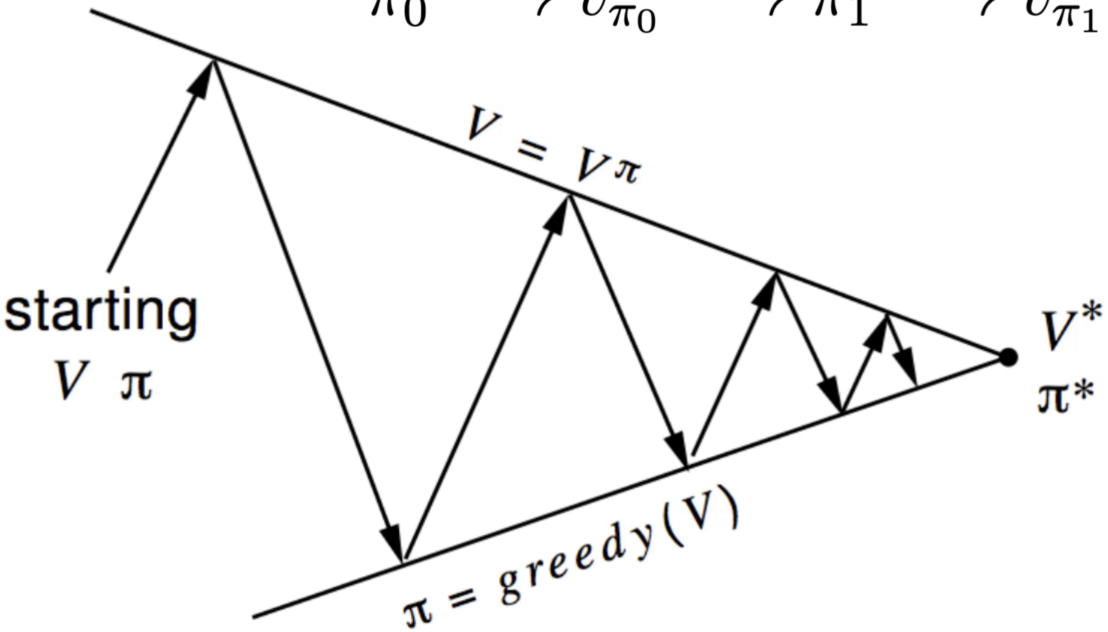
- **步骤 2：策略改进**
  - 在每个状态 s 上，基于 v_π 采取**贪心**行动　$\pi' = \text{greedy}(v_\pi)$

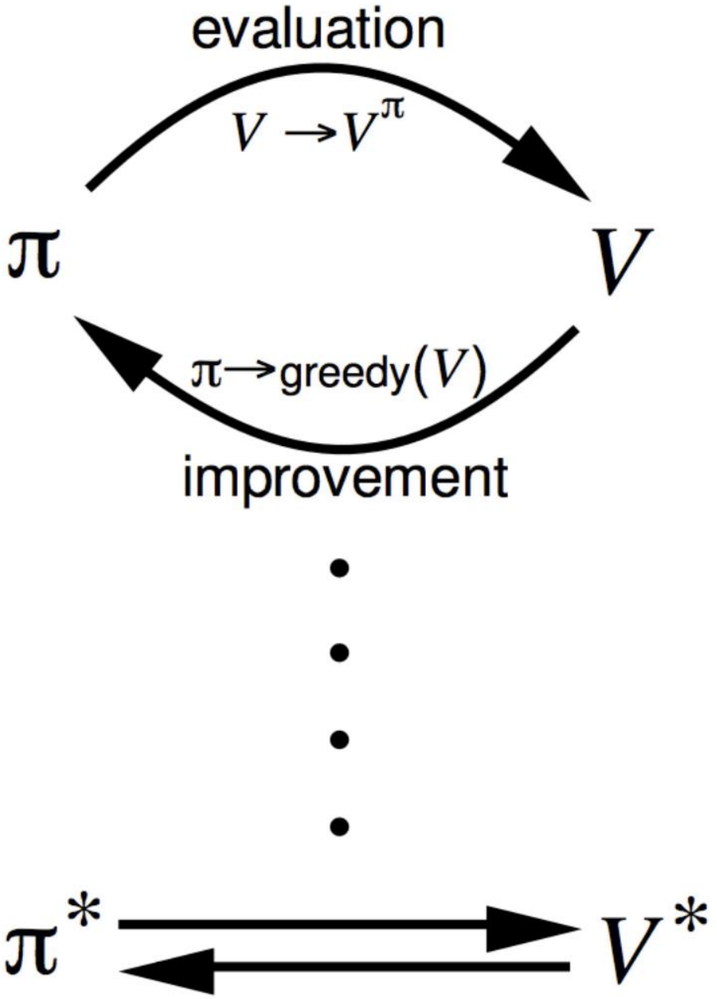
$$\begin{aligned}
\pi'(s) &= \arg\max_a q_\pi(s, a) \\
&= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \arg\max_a \sum_{s', r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big],
\end{aligned}$$

- 扩展阅读：收缩映射定理（Contraction Mapping Theorem）保证上述迭代过程一定会收敛到唯一的不动点，最优策略 π^*

策略迭代

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$





- **策略评估（Policy Evaluation）**：估计策略 π 的值函数 v_π
  - 采用迭代策略评估的方式
- **策略改进（Policy Improvement）**：生成一个新策略 π'，且 $\pi' \geq \pi$
  - 采用贪婪策略提升

策略迭代算法

1. Initialization

   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation

   Repeat

   $\quad \Delta \leftarrow 0$

   $\quad$ For each $s \in \mathcal{S}$:

   $\quad\quad v \leftarrow V(s)$

   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$

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

   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement

   policy-stable $\leftarrow$ *true*

   For each $s \in \mathcal{S}$:

   $\quad a \leftarrow \pi(s)$

   $\quad \pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$

   $\quad$ If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*

   If *policy-stable*, then stop and return V and π; else go to 2

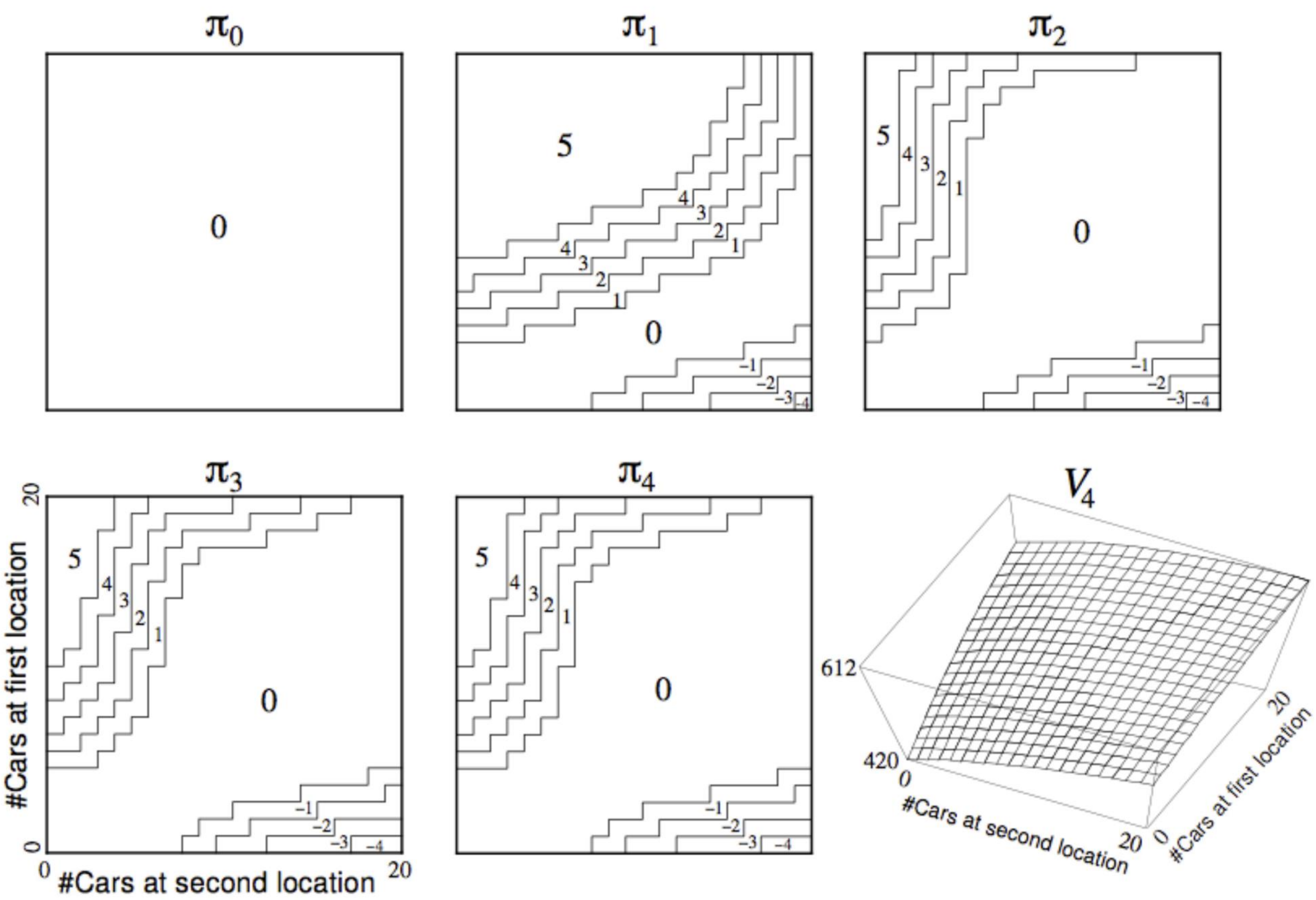
示例：汽车租赁问题

- **状态**：有两个汽车租赁点，每个租赁点最多可停放 20 辆车。
- **行动**：每晚最多可在两个租赁点之间转移 5 辆车。
- **奖励**：每租出一辆车（前提是有车可租）可获得 10 美元奖励。
- **状态转移**：汽车的归还和租用是随机的，服从泊松分布。
  - 出现 n 次归还 / 租用的概率为 $\frac{\lambda^n}{n!}e^{-\lambda}$
  - 第一个租赁点：平均租用次数为 3 次（λ_1），平均归还次数为 3 次（λ_2）。
  - 第二个租赁点：平均租用次数为 4 次（λ_3），平均归还次数为 2 次（λ_4）。

汽车租赁问题：策略迭代算法

```
# 简化伪代码：策略迭代框架
function policy_iteration():
    # 初始化
    定义最大车辆数 max_cars = 20
    定义最大转移数 max_transfer = 5
    初始化值函数 V(s1, s2) = 0(对所有状态)
    初始化策略 π(s1, s2) = 0(初始策略:不转移车辆)

    # 策略迭代循环
    while True:
        # --- 策略评估：计算当前策略的值函数 ---
        重复更新 V(s1, s2) 直到收敛：
            对每个状态 (s1, s2):
                # 根据策略 π 转移车辆
                a = π(s1, s2)
                new_s1 = s1 − a(转移后租赁点1的车辆)
                new_s2 = s2 + a(转移后租赁点2的车辆)

                # 计算期望奖励和下一状态（考虑租用和归还的泊松分布）
                总期望奖励 = Σ [泊松概率 * 实际租出量 * 10]
                总期望未来价值 = Σ [泊松概率 * V(下一状态)]
                V(s1, s2) = 总期望奖励 + 总期望未来价值

        # --- 策略改进：寻找更优动作 ---
        策略稳定标志 = True
        对每个状态 (s1, s2):
            原动作 = π(s1, s2)
            最佳动作 = None
            最佳价值 = −∞

            # 遍历所有可能的转移动作（−5到5）
            for a in [−5, −4, ..., 5]:
                if 转移非法（如车辆不足）: continue

                # 计算动作 a 的期望价值
                当前价值 = 计算转移后的总期望（奖励 + 未来价值）
                if 当前价值 > 最佳价值:
                    最佳价值 = 当前价值
                    最佳动作 = a

            # 更新策略
            π(s1, s2) = 最佳动作
            if 最佳动作 ≠ 原动作:
                策略稳定标志 = False

        # 终止条件：策略不再变化
        if 策略稳定标志:
            break

    return V, π
```

汽车租赁问题



- 正数表示：每晚从 first location 移动到 second location 的车辆数目
- 负数表示：每晚从 second location 移动到 first location 的车辆数目
- π_4 为最优策略

策略提升

- 考虑一个**确定性**策略 $a = \pi(s)$

  我们可以通过采取**贪心策略**来改进这个策略。即：$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ q_\pi(s, a)$

- 这使得从任意状态 s 出发，经过一步后的价值得到提升，即：

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

  因此，这也提升了价值函数，即 $v_{\pi'}(s) \geq v_\pi(s)$

- 具体推导如下：

$$\begin{aligned} v_\pi(s) \leq q_\pi(s, \pi'(s)) &= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \ldots \mid S_t = s\right] = v_{\pi'}(s) \end{aligned}$$

- 其中，$q_\pi(s, a)$ 表示在策略 π 下，从状态 s 采取动作 a 后的动作价值函数 $v_\pi(s)$ 表示在策略 π 下，状态 s 的价值函数；R_t 表示在时刻 t 获得的奖励；S_t 表示在时刻 t 的状态；

- $\mathbb{E}_{\pi'}[\cdot]$ 表示在策略π下的期望。上述推导过程表明，通过贪心策略得到的新策略 π ' 的价值函数不小于原策略π的价值函数

策略提升

- 当策略无法再改进

$$q_{\pi}(s,\pi'(s)) = \max_{a\in\mathcal{A}} q_{\pi}(s,a) = q_{\pi}(s,\pi(s)) = v_{\pi}(s)$$

- 此时满足贝尔曼最优方程

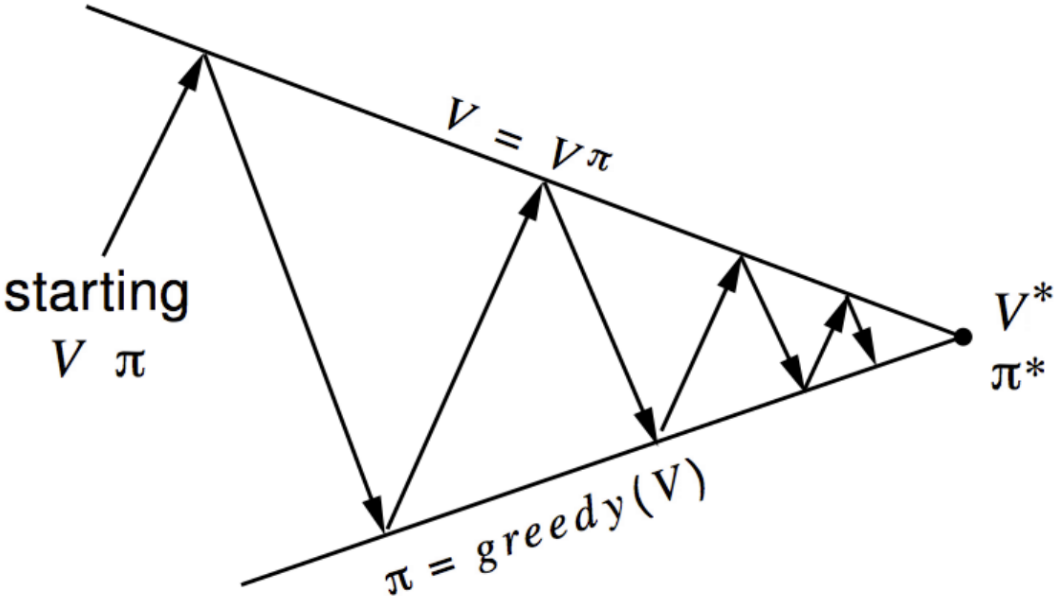
$$v_{\pi}(s) = \max_{a\in\mathcal{A}} q_{\pi}(s,a)$$

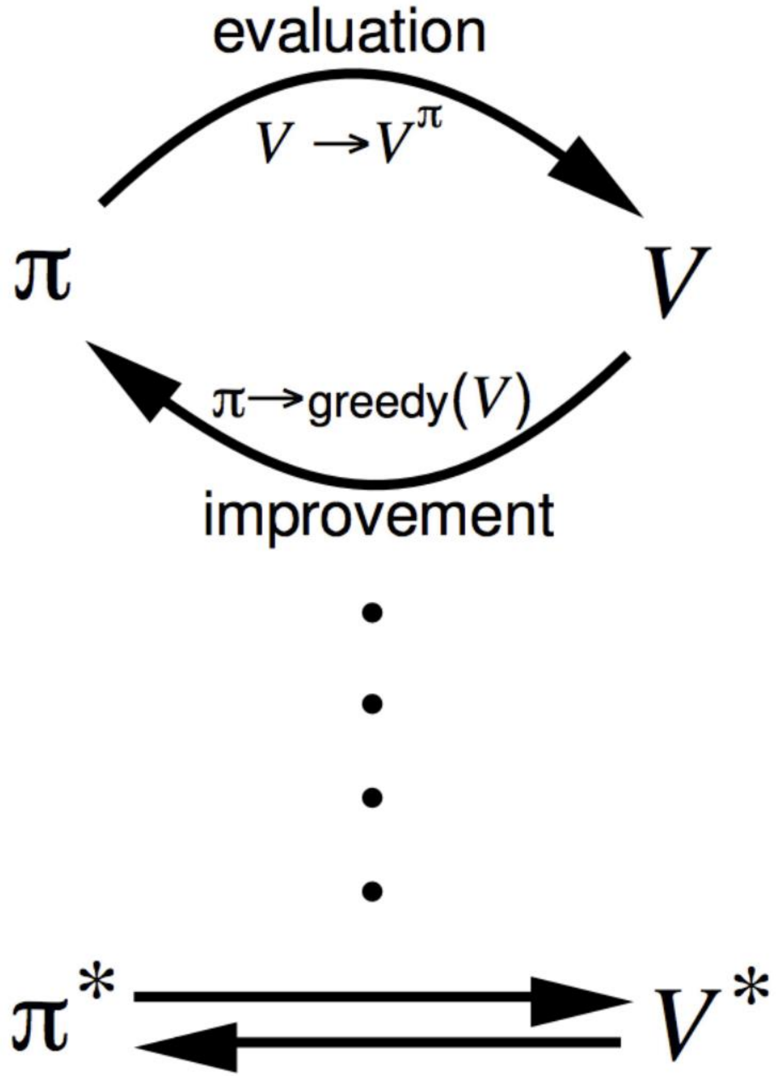
- 则：

$$v_{\pi}(s) = v_{*}(s) \text{ for all } s \in \mathcal{S}$$

- 因而当前策略 π 已经是**最优策略**

广义策略迭代





- **策略评估**：估计策略 π 的值函数 v_π
  - 策略评估不需要精确计算出 v_π
  - **任何**策略评估算法
- **策略改进**：生成一个新策略 π'，且 $\pi' \geq \pi$
  - 策略改进也不局限于贪心算法
  - **任何**策略提升算法

目录

01 动态规划算法：介绍

02 动态规划算法：策略评估

03 动态规划算法：策略迭代

04 动态规划算法：价值迭代

价值迭代

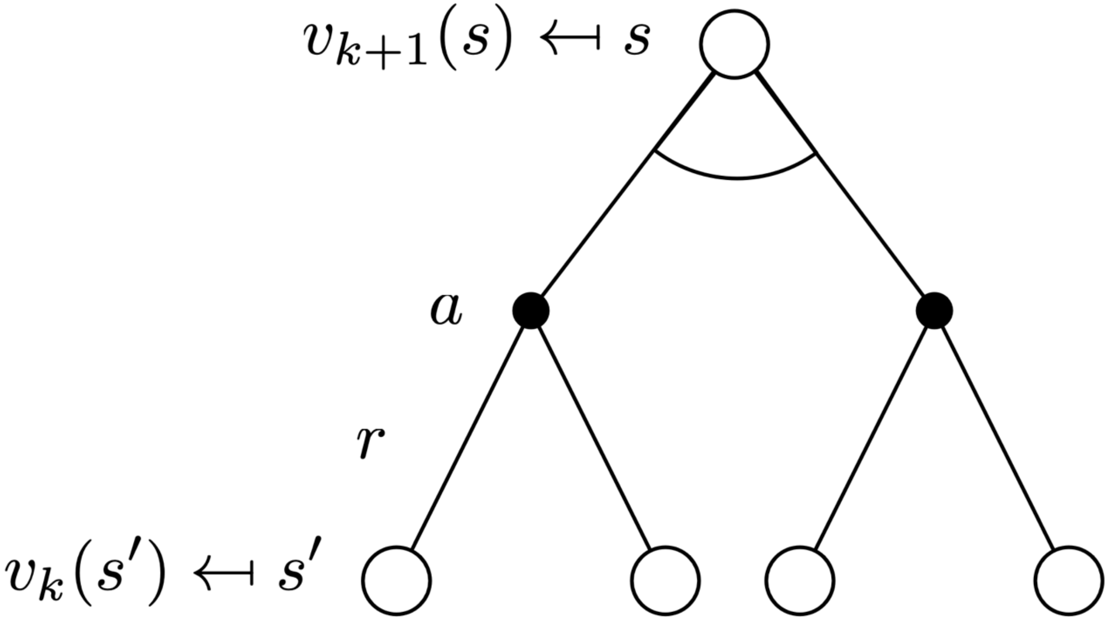
- 若已知子问题的最优值函数解 $v_*(s')$
- 则状态 s 的最优值函数解 $v_*(s)$ 可通过一步迭代得到：

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

$$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_*$$

- 价值迭代：按上述公式迭代更新
- 与**策略迭代**不同，值迭代中没有显式的策略表示， 中间过程的值函数可能不对应任何实际策略

价值迭代



$$\mathbf{v}_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \mathbf{v}_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \boldsymbol{\mathcal{R}}^{\boldsymbol{a}} + \gamma \boldsymbol{\mathcal{P}}^{\boldsymbol{a}} \mathbf{v}_k$$

价值迭代算法

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$

网格世界：价值迭代算法

```
# 伪代码：最短路径问题的价值迭代
function value_iteration():
    # 初始化
    定义状态集合 S(如网格中的坐标或节点)
    定义终点状态 terminal_state
    定义动作集合 A(如上下左右移动)
    定义动作代价 cost = -1(每移动一步的即时奖励)
    折扣因子 γ = 1(无折扣,因为目标是总代价最小化)
    收敛阈值 θ = 0.0001
    初始化值函数 V(s) = 0(对所有状态 s ∈ S)
    V(terminal_state) = 0  # 终点值固定为0

    # 迭代直到收敛
    while True:
        Δ = 0
        创建临时值函数 V_new 的副本（初始化为 V 的当前值）

        # 遍历所有非终止状态
        for 每个状态 s in S(s ≠ terminal_state):
            max_value = -∞
            # 遍历所有可能动作
            for 每个动作 a in A:
                # 获取确定性转移的下一个状态 s'
                s_prime = get_next_state(s, a)
                # 计算新值（奖励为固定动作代价）
                value = cost + γ * V(s_prime)
                if value > max_value:
                    max_value = value
            # 更新临时值函数
            V_new(s) = max_value
            Δ = max(Δ, abs(V(s) - V_new(s)))

        # 同步更新所有状态的值
        V = V_new.copy()

        # 检查收敛条件
        if Δ < θ:
            break

    # 提取最优策略（可选）
    初始化策略 π(s) 为空字典
    for 每个状态 s in S:
        best_action = None
        best_value = -∞
        for 每个动作 a in A:
            s_prime = get_next_state(s, a)
            value = cost + γ * V(s_prime)
            if value > best_value:
                best_value = value
                best_action = a
        π(s) = best_action

    return V, π

# 辅助函数：根据动作返回下一个状态（需根据具体问题实现）
function get_next_state(s, a):
    # 示例：网格世界中移动逻辑
    if a == 'up':
        return (s.x, s.y-1) if 不越界 else s
    elif a == 'down':
        return (s.x, s.y+1) if 不越界 else s
    # ...类似处理其他动作
```

示例：网格世界中的最短路径

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

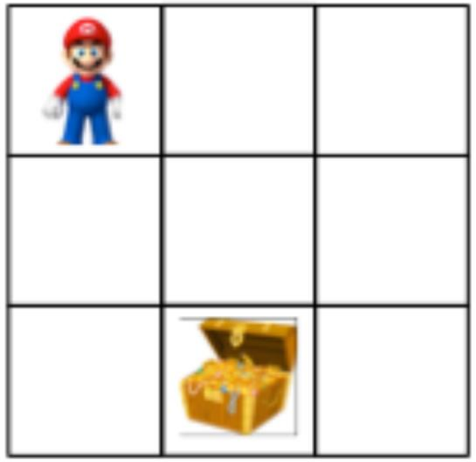
V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

示例：宝箱世界

- 假设我们有一个3 x 3的棋盘：
  - 有一个单元格是超级玛丽，每回合可以往上、下、左、右四个方向移动
  - 有一个单元格是宝藏，超级玛丽找到宝藏则游戏结束，目标是让超级玛丽以最快的速度找到宝藏
  - 假设游戏开始时，宝藏的位置一定是(1, 2)



- 这个一个标准的马尔科夫决策过程(MDP)：
  - 状态空间：超级玛丽当前的坐标
  - 决策空间：上、下、左、右四个动作
  - 奖励函数：
    - 超级玛丽每移动一步，reward = -1
    - 超级玛丽得到宝箱，reward = 0 并且游戏结束

宝箱世界：价值迭代

首先我们定义超级玛丽当前位置的价值 V(state)：从当前 state = (x, y) 开始，能够获得的最大化 Reward 的总和

0	0	0
0	0	0
0	0	0

初始化

-1	-1	-1
-1	-1	-1
-1	0	-1

第一轮迭代

-2	-2	-2
-2	-1	-2
-1	0	-1

第二轮迭代

-3	-2	-3
-2	-1	-2
-1	0	-1

第三轮迭代

- **初始化**：所有状态的价值 V(s) = 0
- 第一轮迭代：对于每个状态，逐一尝试上、下、左、右四个行动
  - 记录行动带来的奖励，以及新状态 V(s')
  - 选择最优的行动，更新 V(s) = Reward + V(s') = -1 + 0
  - 第一轮结束后，大部分状态都有 V(s) = -1，即从当前位置出发走一步获得 Reward=-1

宝箱世界：价值迭代

- 第二轮迭代：对于每个状态，逐一尝试上、下、左、右四个行动
  记录行动带来的奖励，以及新状态 V(s')
  选择最优的行动，更新 V(s) = Reward + V(s')
    对于宝箱周围的状态，最优的行动是一步到达宝箱，V(s) = Reward + V(s') = -1 + 0
    对于其他状态，所有的行动都是一样的，V(s) = Reward + V(s') = -1 + -1
  - 第二轮结束后，宝箱周围状态的价值保持不变 V(s) = -1，其他状态的价值 V(s) = -2
- 第三轮迭代：对于每个状态，逐一尝试上、下、左、右四个行动
    对于宝箱周围的状态，最优的行动是一步到达宝箱，V(s) = Reward + V(s') = -1 + 0
    对于宝箱两步距离的状态，最优的行动是先一步到达宝箱周边的状态，V(s) = Reward + V(s') = -1 + -1
    对于宝箱三步距离的状态，所有行动都是一样的，V(s) = Reward + V(s') = -1 + -2
- 第四轮迭代：对于每个状态，逐一尝试上、下、左、右四个行动
    对于宝箱周围的状态，最优的行动是一步到达宝箱，V(s) = Reward + V(s') = -1 + 0
    对于宝箱两步距离的状态，最优的行动是先一步到达宝箱周边的状态，V(s) = Reward + V(s') = -1 + -1
    对于宝箱三步距离的状态，最优的行动是所有行动都是一样的（因为是 3x3 的格子并且宝箱位置在（1，2），任意合法行动都使超级玛丽更靠近宝箱），V(s) = Reward + V(s') = -1 + -2
  - 在第四轮迭代中，所有 V(s) 更新前后都没有任何变化，价值迭代已经找到了最优策略
- 上面的迭代过程基于贝尔曼最优方程对每个位置的价值进行更新

$$V_*(s) = max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V_*(s')]$$

动态规划算法总结

问题	贝尔曼方程（Bellman Equation）	**算法分类**
预测问题（Prediction）	贝尔曼期望方程（Bellman Expectation Equation）	迭代策略评估（Iterative Policy Evaluation）
控制问题（Control）	贝尔曼期望方程 + 贪心策略改进（Greedy Policy Improvement）	**策略迭代（Policy Iteration）**
控制问题（Control）	贝尔曼最优性方程（Bellman Optimality Equation）	**价值迭代（Value Iteration）**

算法特性

- 基于**状态值函数** $v_\pi(s)$ 或 $v_*(s)$ ，每次迭代复杂度为 $O(mn^2)$（ m 为动作数，n 为状态数 ）。
- 也可基于**动作值函数** $q_\pi(s,a)$ 或 $q_*(s,a)$，每次迭代复杂度为 $O(m^2n^2)$ 。

课后作业

1. 从均匀随机策略开始，采用策略迭代方法，求解课件中“宝箱世界”问题。

谢谢！