第4章 价值迭代与策略迭代

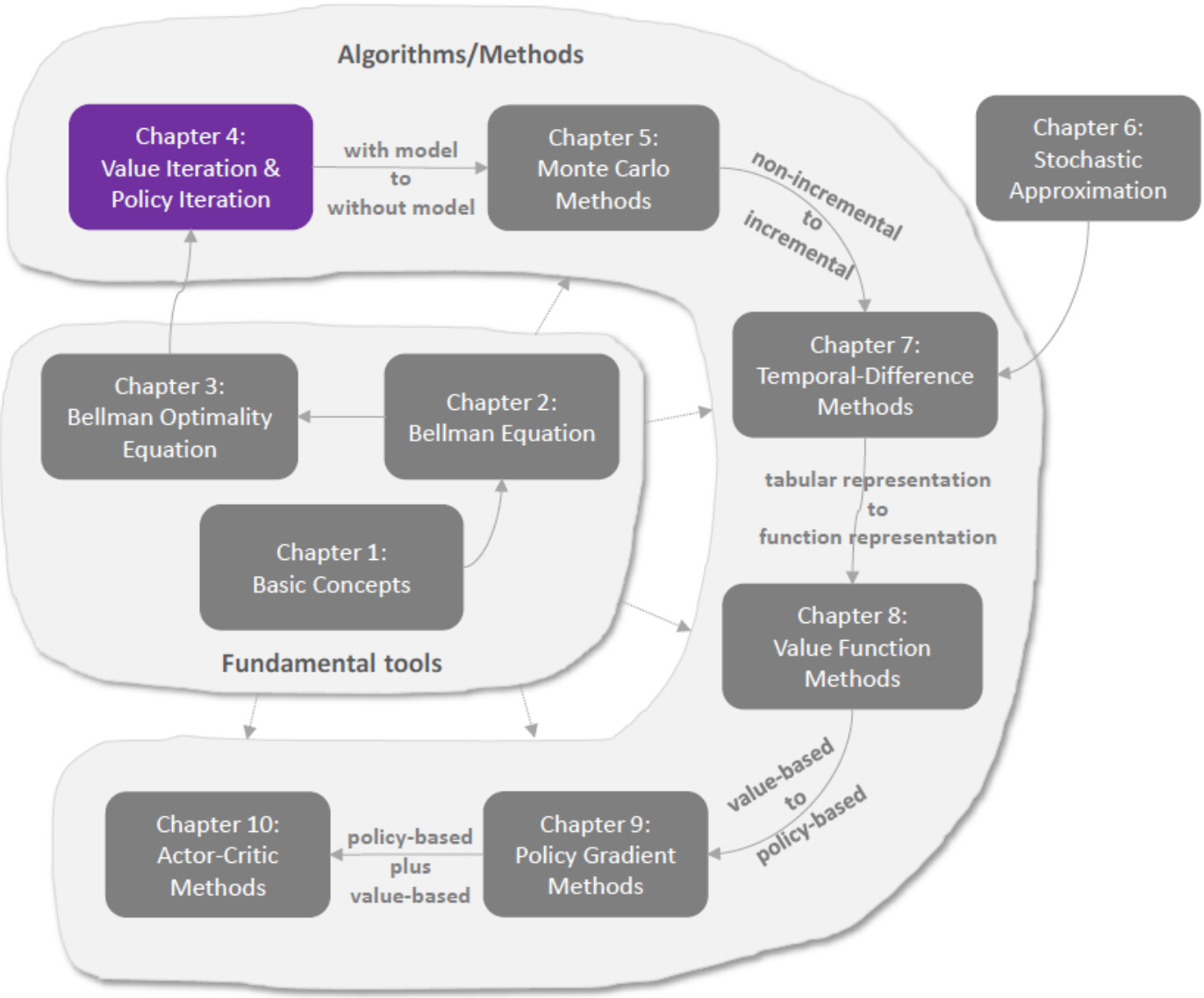

Figure 4.1: Where we are in this book.

图 4.1：我们在本书中的位置。

经过前几章的准备，我们现在准备介绍第一批能够找到最优策略的算法。本章介绍了三种彼此密切相关的算法。

- **第一个是价值迭代（value iteration）算法**，它正是上一章讨论的用于求解贝尔曼最优方程的压缩映射定理所建议的算法。在本章中，我们将更多地关注该算法的实现细节。
- **第二个是策略迭代（policy iteration）算法**，其思想广泛应用于强化学习算法中。

- **第三个是截断策略迭代（truncated policy iteration）算法**，这是一个统一的算法，将价值迭代和策略迭代算法作为特例包含在内。

本章介绍的算法被称为*动态规划*（dynamic programming）算法 [10, 11]，它们需要系统模型。这些算法是后续章节介绍的无模型（model-free）强化学习算法的重要基础。例如，第 5 章介绍的蒙特卡洛算法可以通过扩展本章介绍的策略迭代算法直接获得。

4.1 值迭代

本节介绍***值迭代***（value iteration）算法。它正是上一章（定理 3.3）介绍的由压缩映射定理建议的用于求解贝尔曼最优方程的算法。具体而言，该算法为

$$v_{k+1} = \max_{\pi \in \Pi}(r_\pi + \gamma P_\pi v_k), \quad k = 0, 1, 2, \ldots$$

定理 3.3 保证了当 $k \to \infty$ 时，v_k 和 π_k 分别收敛到最优状态价值和最优策略。

该算法是迭代的，并且在每次迭代中有两个步骤。

- 每次迭代的第一步是***策略更新***（policy update）步骤。在数学上，它的目的是找到一个能够求解以下优化问题的策略：

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_k),$$

其中 v_k **是在前一次迭代中获得的。**

- 第二步称为***价值更新***（value update）步骤。在数学上，它通过下式计算新的价值 v_{k+1}：

$$v_{k+1} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k, \tag{4.1}$$

其中 v_{k+1} 将在下一次迭代中使用。

上面介绍的价值迭代算法采用的是矩阵-向量形式。为了实现该算法，我们需要进一步检查其元素形式。虽然矩阵-向量形式有助于理解算法的核心思想，但元素形式对于解释实现细节是必要的。

4.1.1 元素形式与实现

考虑时间步 k 和状态 s 。

- 首先，策略更新步骤 $\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_k)$ 的元素形式为

$$\pi_{k+1}(s) = \arg\max_{\pi} \sum_a \pi(a|s) \underbrace{\left(\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_k(s') \right)}_{q_k(s,a)}, \quad s \in \mathcal{S}.$$

我们在 3.3.1 节中已经表明，能够解决上述优化问题的最优策略是

$$\pi_{k+1}(a|s) = \begin{cases} 1, & a = a_k^*(s), \\ 0, & a \neq a_k^*(s), \end{cases} \tag{4.2}$$

其中 $a_k^*(s) = \arg\max_a q_k(s,a)$。如果 $a_k^*(s) = \arg\max_a q_k(s,a)$ 有多个解，我们可以选择其中任何一个，而不会影响算法的收敛性。由于新策略 π_{k+1} 选择具有最大 $q_k(s,a)$ 的动作，这种策略被称为*贪婪*（greedy）策略。

- 其次，价值更新步骤 $v_{k+1} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k$ 的元素形式为

$$v_{k+1}(s) = \sum_a \pi_{k+1}(a|s) \underbrace{\left(\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_k(s') \right)}_{q_k(s,a)}, \quad s \in \mathcal{S}.$$

将 (4.2) 代入上述方程可得

$$v_{k+1}(s) = \max_a q_k(s,a).$$

总之，上述步骤可以图示为

$$v_k(s) \to q_k(s,a) \to \text{新贪婪策略 } \pi_{k+1}(s) \to \text{新价值 } v_{k+1}(s) = \max_a q_k(s,a)$$

实现细节总结在算法 4.1 中。

一个可能令人困惑的问题是，(4.1) 中的 v_k 是否是状态价值。答案是否定的。虽然 v_k 最终会收敛到最优状态价值，但不能保证它满足任何策略的贝尔曼方程。例如，通常情况下它不满足 $v_k = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k$ 或 $v_k = r_{\pi_k} + \gamma P_{\pi_k} v_k$。它仅仅是算法生成的中间值。此外，由于 v_k 不是状态价值，因此 q_k 也不是动作价值。

算法 4.1：价值迭代算法

📌 初始化：概率模型 $p(r|s,a)$ 和 $p(s'|s,a)$ 对于所有 (s,a) 均已知。初始猜测 v_0。

目标：搜索最优状态价值和最优策略以求解贝尔曼最优方程。

当 v_k 尚未收敛（即 $\|v_k - v_{k-1}\|$ 大于预定义的微小阈值）时，对于第 k 次迭代，执行以下操作：

对于每个状态 $s \in \mathcal{S}$，做

对于每个动作 $a \in \mathcal{A}(s)$，做

q-value：$q_k(s,a) = \sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_k(s')$

最大动作价值：$a_k^*(s) = \arg\max_a q_k(s,a)$

策略更新：如果 $a = a_k^*$，则 $\pi_{k+1}(a|s) = 1$，否则 $\pi_{k+1}(a|s) = 0$

价值更新：$v_{k+1}(s) = \max_a q_k(s,a)$

4.1.2 说明性示例

接下来，我们给出一个例子来说明价值迭代算法的逐步实现。这个例子是一个 2x2 的网格，其中有一个禁止区域

表 4.1：图 4.2 所示示例的 $q(s,a)$ 表达式。

q-table	a_1	a_2	a_3	a_4	a_5
s_1	$-1+\gamma v(s_1)$	$-1+\gamma v(s_2)$	$0+\gamma v(s_3)$	$-1+\gamma v(s_1)$	$0+\gamma v(s_1)$
s_2	$-1+\gamma v(s_2)$	$-1+\gamma v(s_2)$	$1+\gamma v(s_4)$	$0+\gamma v(s_1)$	$-1+\gamma v(s_2)$
s_3	$0+\gamma v(s_1)$	$1+\gamma v(s_4)$	$-1+\gamma v(s_3)$	$-1+\gamma v(s_3)$	$0+\gamma v(s_3)$
s_4	$-1+\gamma v(s_2)$	$-1+\gamma v(s_4)$	$-1+\gamma v(s_4)$	$0+\gamma v(s_3)$	$1+\gamma v(s_4)$

目标区域是 s_4。奖励设置是 $r_{\text{boundary}} = r_{\text{forbidden}} = -1$ 且 $r_{\text{target}} = 1$。折扣率是 $\gamma = 0.9$。

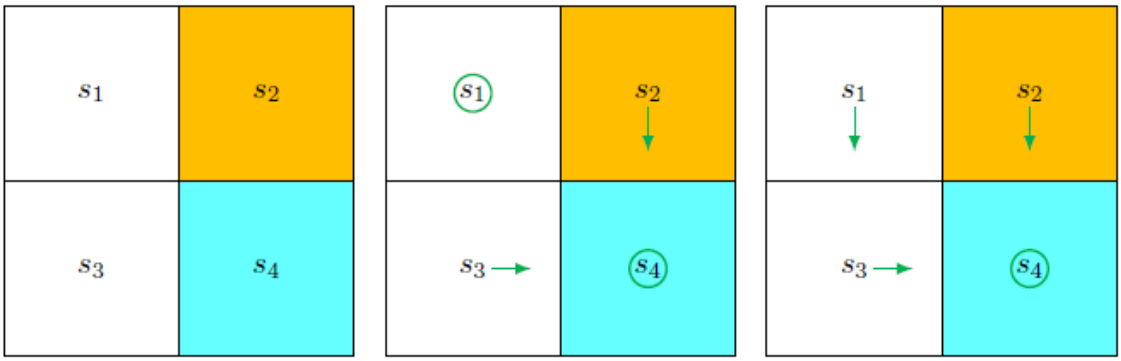

Figure 4.2: An example for demonstrating the implementation of the value iteration algorithm.

图 4.2：演示价值迭代算法实现的示例。

每个状态-动作对的 *q-value*表达式如表 4.1 所示。

- $k = 0$:

  不失一般性，选择初始值为 $v_0(s_1) = v_0(s_2) = v_0(s_3) = v_0(s_4) = 0$。

  q-value计算：将 $v_0(s_i)$ 代入表 4.1 给出了表 4.2 所示的 *q-value*。

q-table	a_1	a_2	a_3	a_4	a_5
s_1	-1	-1	0	-1	0
s_2	-1	-1	1	0	-1
s_3	0	1	-1	-1	0
s_4	-1	-1	-1	0	1

表 4.2：$k = 0$ 时的 $q(s,a)$ 值。

q-table	a_1	a_2	a_3	a_4	a_5
s_1	$-1+\gamma 0$	$-1+\gamma 1$	$0+\gamma 1$	$-1+\gamma 0$	$0+\gamma 0$
s_2	$-1+\gamma 1$	$-1+\gamma 1$	$1+\gamma 1$	$0+\gamma 0$	$-1+\gamma 1$
s_3	$0+\gamma 0$	$1+\gamma 1$	$-1+\gamma 1$	$-1+\gamma 1$	$0+\gamma 1$
s_4	$-1+\gamma 1$	$-1+\gamma 1$	$-1+\gamma 1$	$0+\gamma 1$	$1+\gamma 1$

表 4.3：$k=1$ 时的 $q(s,a)$ 值。

策略更新：π_1 是通过为每个状态选择具有最大 q-值的动作而获得的：

$$\pi_1(a_5|s_1)=1, \quad \pi_1(a_3|s_2)=1, \quad \pi_1(a_2|s_3)=1, \quad \pi_1(a_5|s_4)=1.$$

该策略在图 4.2（中间的子图）中进行了可视化。很明显，这个策略不是最优的，因为它选择在 s_1 处保持静止。值得注意的是，(s_1,a_5) 和 (s_1,a_3) 的 q-value实际上是相同的，我们可以随机选择其中任何一个动作。

价值更新：v_1 是通过将 v-value更新为每个状态的最大 q-value而获得的：

$$v_1(s_1)=0, \quad v_1(s_2)=1, \quad v_1(s_3)=1, \quad v_1(s_4)=1.$$

◇ $k=1$：

q-value计算：将 $v_1(s_i)$ 代入表 4.1 得到了表 4.3 中所示的 q-值。

策略更新：π_2 是通过选择最大的 q-值而获得的：

$$\pi_2(a_3|s_1)=1, \quad \pi_2(a_3|s_2)=1, \quad \pi_2(a_2|s_3)=1, \quad \pi_2(a_5|s_4)=1.$$

该策略在图 4.2（右侧子图）中进行了可视化。

价值更新：v_2 是通过将 v-值更新为每个状态的最大 q-值而获得的：

$$v_2(s_1)=\gamma 1, \quad v_2(s_2)=1+\gamma 1, \quad v_2(s_3)=1+\gamma 1, \quad v_2(s_4)=1+\gamma 1.$$

◇ $k=2,3,4,\ldots$

值得注意的是，如图 4.2 所示，策略 π_2

已经是最优的。因此，我们在这个简单的例子中，只需要运行两次迭代即可获得最优策略。对于更复杂的例子，我们需要运行更多次迭代，直到 v_k 的值收敛（例如，直到 $\|v_{k+1}-v_k\|$ 小于预先指定的阈值）。

4.2 策略迭代 (Policy iteration)

本节将介绍另一种重要的算法：*策略迭代 (policy iteration)*。与价值迭代不同，策略迭代并非用于直接求解贝尔曼最优方程。然而，如下文所示，它与价值迭代有着密切的联系。此外，策略迭代的思想非常重要，因为它在强化学习算法中被广泛应用。

4.2.1 算法分析

策略迭代是一种迭代算法。每次迭代包含两个步骤。

- 第一步是***策略评估 (policy evaluation)*** 步骤。顾名思义，该步骤通过计算相应的状态价值来评估给定的策略。也就是求解以下贝尔曼方程：

$$v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}, \quad (4.3)$$

其中 π_k 是在上一次迭代中获得的策略，v_{π_k} 是待计算的状态价值。r_{π_k} 和 P_{π_k} 的值可以从系统模型中获得。

- 第二步**是*策略改进 (policy improvement)* 步骤**。顾名思义，该步骤用于改进策略。具体来说，一旦在第一步中计算出了 v_{π_k} ，就可以通过下式获得新的策略 π_{k+1} ：

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k}). \quad \text{(这里是更新 } \pi \text{ ,而不是value)}$$

根据上述对算法的描述，自然会引出三个问题：

- 在策略评估步骤中，如何求解状态价值 v_{π_k} ？
- 在策略改进步骤中，为什么新策略 π_{k+1} 比 π_k 更好?
- 为什么该算法最终能收敛到最优策略?

接下来我们将逐一回答这些问题。

在策略评估步骤中，如何计算 v_{π_k} ？

我们在第 2 章中介绍了两种求解式 (4.3) 中贝尔曼方程的方法。接下来我们将简要回顾这两种方法。第一种方法是闭式解 (closed-form solution)：

$$v_{\pi_k} = (I - \gamma P_{\pi_k})^{-1} r_{\pi_k}$$

这种闭式解对于理论分析很有用，但由于需要其他数值算法来计算矩阵逆，因此实现效率较低。第二种方法是一种易于实现的迭代算法：

$$v_{\pi_k}^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}, \quad j = 0, 1, 2, \ldots \quad (4.4)$$

其中 $v_{\pi_k}^{(j)}$ 表示 v_{π_k} 的第 j 次估计值。从任意初始猜测 $v_{\pi_k}^{(0)}$ 开始，可以保证当 $j \to \infty$ 时，$v_{\pi_k}^{(j)} \to v_{\pi_k}$ 。详情请参见第 2.7 节。

有趣的是，策略迭代是一个迭代算法，其策略评估步骤中嵌入了另一个迭代算法 (4.4)。理论上，这个嵌入的迭代算法需要*无限*步（即 $j \to \infty$ ）才能收敛到真实的状态价值 v_{π_k} 。

然而，这在现实中是不可能实现的。在实践中，迭代过程会在满足特定准则时终止。例如，终止准则可以是 $\|v_{\pi_k}^{(j+1)} - v_{\pi_k}^{(j)}\|$ 小于预先指定的阈值，或者是 j 超过预先指定的值。如果我们不运行无限次迭代，我们只能获得 v_{π_k} 的不精确值，该值将用于随后的策略改进步骤。这会导致问题吗？答案是否定的。当我们稍后在 4.3 节介绍截断策略迭代算法时，原因就会变得清晰。

在策略改进步骤中，为什么 π_{k+1} 比 π_k 更好?

策略改进步骤可以改进给定的策略，如下所示。

引理 4.1 (策略改进)。如果 $\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k})$ ，那么 $v_{\pi_{k+1}} \geq v_{\pi_k}$ 。(这一步的目的是证明收敛性，收敛了就可以保证 π_{k+1} **比** π_k)

这里，$v_{\pi_{k+1}} \geq v_{\pi_k}$ 意味着对于所有 s ，都有 $v_{\pi_{k+1}}(s) \geq v_{\pi_k}(s)$。本引理的证明见方框 4.1。

方框 4.1：引理 4.1 的证明

由于 $v_{\pi_{k+1}}$ 和 v_{π_k} 是状态价值，它们满足贝尔曼方程：

$$v_{\pi_{k+1}} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}},$$

$$v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

注：在所有策略 (π) 里，让向量（或逐状态的量），

$$\pi_{k+1} = \arg\max_{\pi} \left(r_\pi + \gamma P_\pi v_{\pi_k} \right)$$

由于 $\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k})$ ，**取最大值的定义，按“最大化”的定义，用最大化得到的那个策略去算，结果一定不小于用任何别的策略去算，特别是不小于取 ($\pi = \pi_k$) 时的值：**

$$r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k} \geq r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

更“逐状态”地看（通常教材默认是逐状态贪心）

上面那个不等式一般是按**分量逐状态**理解的：对每个状态 (s),

$$[\pi_{k+1} + \gamma P_{\pi_{k+1}} v_{\pi_k}](s) \geq [\pi_k + \gamma P_{\pi_k} v_{\pi_k}](s)$$

把它展开就是：

$$r_{\pi_{k+1}}(s) + \gamma \sum_{s'} P_{\pi_{k+1}}(s, s'), v_{\pi_k}(s') \geq r_{\pi_k}(s) + \gamma \sum_{s'} P_{\pi_k}(s, s'), v_{\pi_k}(s').$$

直观上：(π_{k+1}) 在每个状态都选一个让**“即时奖励 + 折扣后的下一步价值”**最大的动作，所以当然不会比 (π_k) 在该状态下原来选的动作更差。

由此可得

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} &= (r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}) - (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) - (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq \gamma P_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}). \end{aligned}$$

因此，

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} &\leq \gamma^2 P_{\pi_{k+1}}^2 (v_{\pi_k} - v_{\pi_{k+1}}) \leq \cdots \leq \gamma^n P_{\pi_{k+1}}^n (v_{\pi_k} - v_{\pi_{k+1}}) \\ &\leq \lim_{n \to \infty} \gamma^n P_{\pi_{k+1}}^n (v_{\pi_k} - v_{\pi_{k+1}}) = 0. \end{aligned}$$

该极限为 0 是基于以下事实：当 $n \to \infty$ 时 $\gamma^n \to 0$ ，并且对于任意 n ， $P_{\pi_{k+1}}^n$ 都是非负随机矩阵。这里，随机矩阵指的是行和全为 1 的非负矩阵。

为什么策略迭代算法最终能找到最优策略?

策略迭代算法生成两个序列。

第一个是策略序列： $\{\pi_0, \pi_1, \ldots, \pi_k, \ldots\}$ 。第二个是状态价值序列： $\{v_{\pi_0}, v_{\pi_1}, \ldots, v_{\pi_k}, \ldots\}$ 。假设 v^* 是最优状态价值。那么，对于所有 k ，都有 $v_{\pi_k} \leq v^*$ 。由于根据引理 4.1 策略在不断改进，我们知道

$$v_{\pi_0} \leq v_{\pi_1} \leq v_{\pi_2} \leq \cdots \leq v_{\pi_k} \leq \cdots \leq v^*.$$

由于 v_{π_k} 是非递减的且总是有上界 v^*，根据单调收敛定理 [12]（附录 C），当 $k \to \infty$ 时，v_{π_k} 收敛到一个常数值，记为 v_∞。接下来的分析表明 $v_\infty = v^*$。

定理 4.1（策略迭代的收敛性）。由策略迭代算法生成的状态价值序列 $\{v_{\pi_k}\}_{k=0}^{\infty}$ 收敛于最优状态价值 v^*。结果是，策略序列 $\{\pi_k\}_{k=0}^{\infty}$ 收敛于最优策略。

该定理的证明见方框 4.2。该证明不仅展示了策略迭代算法的收敛性，还揭示了策略迭代与价值迭代算法之间的关系。通俗地说，如果两种算法从相同的初始猜测开始，策略迭代将比价值迭代收敛得更快，这是因为策略评估步骤中嵌入了额外的迭代。当我们在 4.3 节介绍截断策略迭代算法时，这一点将变得更加清晰。

> **方框 4.2：定理 4.1 的证明**
>
> 证明的思路是展示策略迭代算法比价值迭代算法收敛得更快。
>
> 特别地，为了证明 $\{v_{\pi_k}\}_{k=0}^{\infty}$ 的收敛性，我们引入另一个由下式生成的序列 $\{v_k\}_{k=0}^{\infty}$：
>
> $$v_{k+1} = f(v_k) = \max_{\pi}(r_\pi + \gamma P_\pi v_k).$$ 这里最大化后，求的是 v_k
>
> 这个迭代算法正是价值迭代算法。我们已经知道，对于给定的任意初始值 v_0，v_k 都会收敛到 v^*。
>
> 对于 $k = 0$，我们总能找到一个 v_0 使得对于任意 π_0 都有 $v_{\pi_0} \geq v_0$。
>
> 接下来我们通过归纳法证明对于所有 k，都有 $v_k \leq v_{\pi_k} \leq v^*$。
>
> 对于 $k \geq 0$，假设 $v_{\pi_k} \geq v_k$。
>
> 对于 $k+1$，我们有
>
> $$\begin{aligned}
> v_{\pi_{k+1}} - v_{k+1} &= (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) - \max_{\pi}(r_\pi + \gamma P_\pi v_k) \\
> &\geq (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) - \max_{\pi}(r_\pi + \gamma P_\pi v_k) \\
> &\quad (\text{因为根据引理 4.1 有 } v_{\pi_{k+1}} \geq v_{\pi_k} \text{ 且 } P_{\pi_{k+1}} \geq 0) \\
> &= (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) - (r_{\pi'_k} + \gamma P_{\pi'_k} v_k) \\
> &\quad (\text{假设 } \pi'_k = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_k)) \\
> &\geq (r_{\pi'_k} + \gamma P_{\pi'_k} v_{\pi_k}) - (r_{\pi'_k} + \gamma P_{\pi'_k} v_k) \\
> &\quad (\text{因为 } \pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k})) \\
> &= \gamma P_{\pi'_k}(v_{\pi_k} - v_k).
> \end{aligned}$$
>
> 由于 $v_{\pi_k} - v_k \geq 0$ 且 $P_{\pi'_k}$ 是非负的，我们有 $P_{\pi'_k}(v_{\pi_k} - v_k) \geq 0$，因此 $v_{\pi_{k+1}} - v_{k+1} \geq 0$。
>
> 因此，我们可以通过归纳法证明对于任意 $k \geq 0$ 都有 $v_k \leq v_{\pi_k} \leq v^*$。由于 v_k 收敛到 v^*，所以 v_{π_k} 也收敛到 v^*。

4.2.2 逐元素形式与实现 (Elementwise form and implementation)

为了实现策略迭代算法，我们需要研究其逐元素形式。

- 首先，策略评估步骤通过使用式 (4.4) 中的迭代算法求解 v_{π_k} ，公式为 $v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}$ 。该算法的逐元素形式为

$$v_{\pi_k}^{(j+1)}(s) = \sum_a \pi_k(a|s) \left(\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}^{(j)}(s') \right), \quad s \in \mathcal{S},$$

其中 $j = 0, 1, 2, \ldots$ 。

- 第二，策略改进步骤求解 $\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k})$ 。该方程的逐元素形式为

$$\pi_{k+1}(s) = \arg\max_{\pi} \sum_a \pi(a|s) \underbrace{\left(\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s') \right)}_{q_{\pi_k}(s,a)}, \quad s \in \mathcal{S},$$

其中 $q_{\pi_k}(s,a)$ 是策略 π_k 下的动作价值。令 $a_k^*(s) = \arg\max_a q_{\pi_k}(s,a)$ 。那么，贪婪最优策略为(在贪婪策略下，最大化动作价值的概率为1)

$$\pi_{k+1}(a|s) = \begin{cases} 1, & a = a_k^*(s), \\ 0, & a \neq a_k^*(s). \end{cases}$$

实现细节总结在算法 4.2 中。

> **算法 4.2：策略迭代算法**
>
> 初始化：已知系统模型 $p(r|s,a)$ 和 $p(s'|s,a)$ ，对于所有 (s,a) 。初始猜测 π_0 。
>
> 目标：搜索最优状态价值和最优策略。
>
> 当 v_{π_k} 未收敛时，进行第 k 次迭代，做
>
> **策略评估：**
>
> 初始化：任意初始猜测 $v_{\pi_k}^{(0)}$
>
> 当 $v_{\pi_k}^{(j)}$ 未收敛时，进行第 j 次迭代，做
>
> 对于每一个状态 $s \in \mathcal{S}$ ，做
>
> $$v_{\pi_k}^{(j+1)}(s) = \sum_a \pi_k(a|s) \left[\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}^{(j)}(s') \right]$$
>
> **策略改进：**
>
> 对于每一个状态 $s \in \mathcal{S}$ ，做
>
> 对于每一个动作 $a \in \mathcal{A}$ ，做
>
> $$q_{\pi_k}(s,a) = \sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s')$$
>
> $$a_k^*(s) = \arg\max_a q_{\pi_k}(s,a)$$
>
> 如果 $a = a_k^*$ ，则 $\pi_{k+1}(a|s) = 1$ ，否则 $\pi_{k+1}(a|s) = 0$

4.2.3 说明性示例 (Illustrative examples)

一个简单的例子

考虑图 4.3 所示的一个简单例子。有两个状态和三个可能的动作：$\mathcal{A}=\{a_\ell, a_0, a_r\}$。这三个动作分别代表向左移动、保持不变和向右移动。奖励设置为 $r_{\text{boundary}}=-1$ 和 $r_{\text{target}}=1$。折扣率为 $\gamma=0.9$。

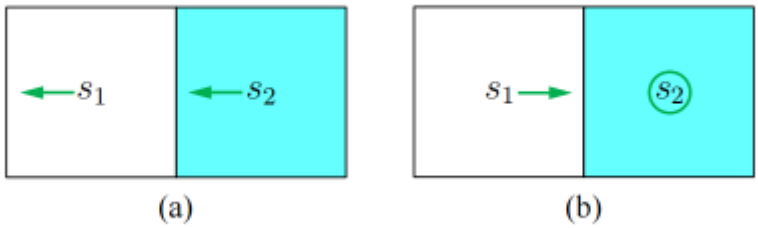

图 4.3：用于说明策略迭代算法实现的例子。

接下来我们将逐步展示策略迭代算法的实现。当 $k=0$ 时，我们从图 4.3(a) 所示的初始策略开始。这个策略并不好，因为它没有向目标区域移动。接下来我们要展示如何应用策略迭代算法来获得最优策略。

- 首先，在策略评估步骤中，我们需要求解贝尔曼方程：

$$v_{\pi_0}(s_1)=-1+\gamma v_{\pi_0}(s_1),$$
$$v_{\pi_0}(s_2)=0+\gamma v_{\pi_0}(s_1).$$

由于方程很简单，可以手动求解得到

$$v_{\pi_0}(s_1)=-10, \quad v_{\pi_0}(s_2)=-9.$$

在实践中，该方程可以通过式 (4.4) 中的迭代算法求解。例如，选择初始状态价值为 $v_{\pi_0}^{(0)}(s_1)=v_{\pi_0}^{(0)}(s_2)=0$。由式 (4.4) 可得

$$\begin{cases} v_{\pi_0}^{(1)}(s_1)=-1+\gamma v_{\pi_0}^{(0)}(s_1)=-1, \\ v_{\pi_0}^{(1)}(s_2)=0+\gamma v_{\pi_0}^{(0)}(s_1)=0, \\ v_{\pi_0}^{(2)}(s_1)=-1+\gamma v_{\pi_0}^{(1)}(s_1)=-1.9, \\ v_{\pi_0}^{(2)}(s_2)=0+\gamma v_{\pi_0}^{(1)}(s_1)=-0.9, \\ v_{\pi_0}^{(3)}(s_1)=-1+\gamma v_{\pi_0}^{(2)}(s_1)=-2.71, \\ v_{\pi_0}^{(3)}(s_2)=0+\gamma v_{\pi_0}^{(2)}(s_1)=-1.71, \\ \vdots \end{cases}$$

随着迭代次数的增加，我们可以看到趋势：随着 j 的增加

$$v_{\pi_0}^{(j)}(s_1)\to v_{\pi_0}(s_1)=-10$$
$$v_{\pi_0}^{(j)}(s_2)\to v_{\pi_0}(s_2)=-9。$$

- 第二，在策略改进步骤中，关键是为每个状态-动作对计算 $q_{\pi_k}(s,a)$。下面的 q 表（q-table）可以用来演示这一过程：

$q_{\pi_k}(s,a)$	a_ℓ	a_0	a_r
s_1	$-1+\gamma v_{\pi_k}(s_1)$	$0+\gamma v_{\pi_k}(s_1)$	$1+\gamma v_{\pi_k}(s_2)$
s_2	$0+\gamma v_{\pi_k}(s_1)$	$1+\gamma v_{\pi_k}(s_2)$	$-1+\gamma v_{\pi_k}(s_2)$

Table 4.4: The expression of $q_{\pi_k}(s,a)$ for the example in Figure 4.3.

表 4.4：图 4.3 中示例的 $q_{\pi_k}(s,a)$ 表达式。

将在上一步策略评估中获得的 $v_{\pi_0}(s_1)=-10, v_{\pi_0}(s_2)=-9$ 代入表 4.4，得到表 4.5。

$q_{\pi_0}(s,a)$	a_ℓ	a_0	a_r
s_1	-10	-9	-7.1
s_2	-9	-7.1	-9.1

Table 4.5: The value of $q_{\pi_k}(s,a)$ when $k=0$.

表 4.5：当 $k=0$ 时 $q_{\pi_k}(s,a)$ 的值，这个表很有意思，即使是概率为0的情况的 $q_{\pi_k}(s,a)$ 也需要求解

通过寻找 q_{π_0} 的最大值，可以获得改进后的策略 π_1 为

$$\pi_1(a_r|s_1)=1, \quad \pi_1(a_0|s_2)=1.$$

该策略如图 4.3(b) 所示。很明显，该策略是最优的。

上述过程表明，在这个简单的例子中，单次迭代足以找到最优策略。对于更复杂的例子，则需要更多次迭代。

一个更复杂的例子 (A more complicated example)

接下来，我们将使用图 4.4 所示的一个更复杂的例子来演示策略迭代算法。奖励设置如下：$r_{\text{boundary}}=-1$，$r_{\text{forbidden}}=-10$，以及 $r_{\text{target}}=1$。折扣率为 $\gamma=0.9$。当从随机初始策略（图 4.4(a)）开始时，策略迭代算法可以收敛到最优策略（图 4.4(h)）。

在迭代过程中观察到了两个有趣的现象。

- 第一，如果我们观察策略是如何演变的，会发现一个有趣的模式：**靠近目标区域的状态比远离目标区域的状态更早找到最优策略。只有当靠近的状态能够先找到通往目标的轨迹时，较远的状态才能找到穿过这些靠近状态到达目标的轨迹。**
- 第二，状态价值的空间分布表现出一个有趣的模式：**位于离目标更近的状态具有更大的状态价值。**这种模式的原因在于，**从较远状态出发的智能体必须移动许多步才能获得正奖励。这些奖励会被严重折现，因此相对较小。**

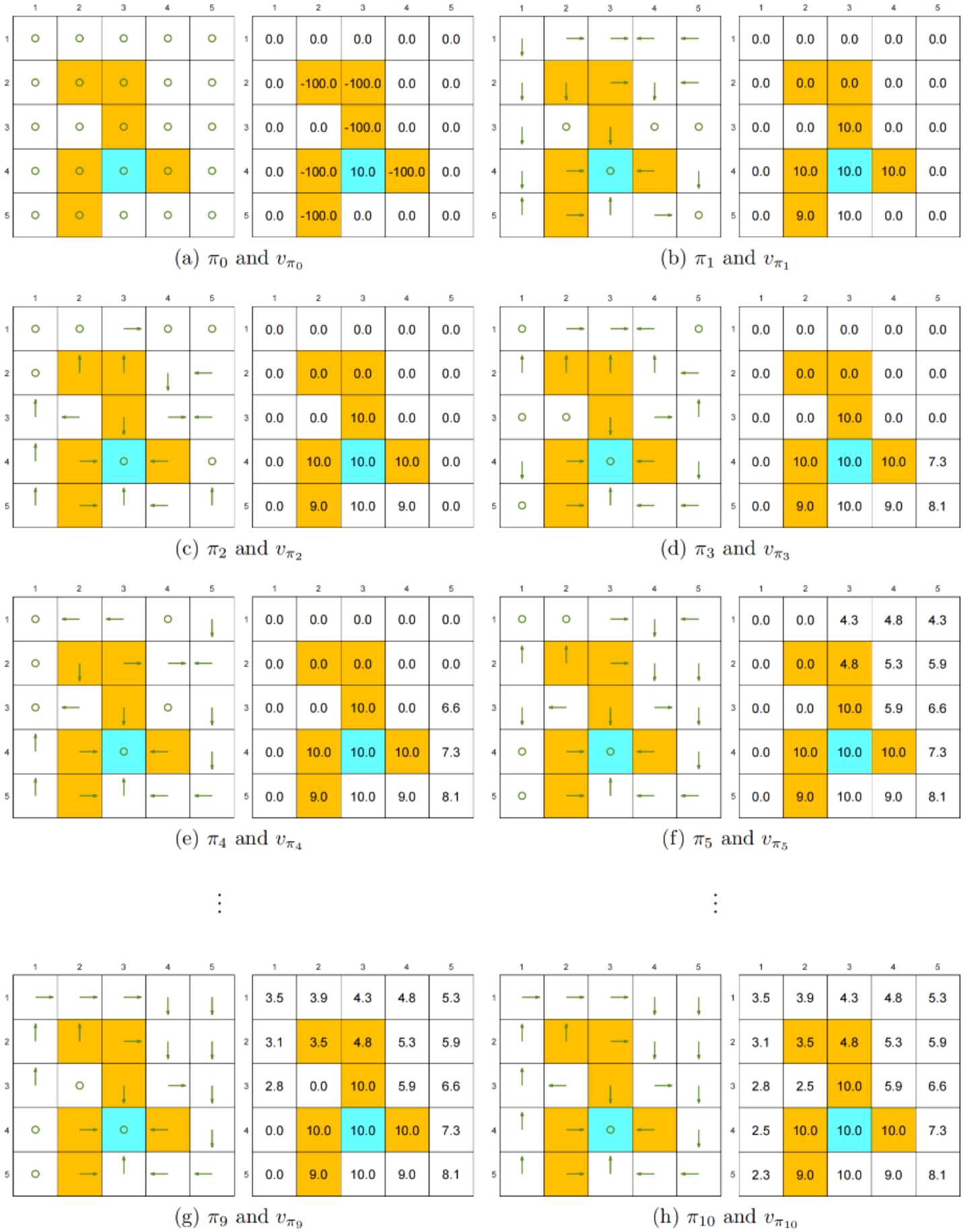

Figure 4.4: The evolution processes of the policies generated by the policy iteration algorithm.

4.3 截断策略迭代 (Truncated policy iteration)

接下来我们介绍一种更通用的算法，称为*截断策略迭代 (truncated policy iteration)*。我们将看到，价值迭代和策略迭代算法是截断策略迭代算法的两个特例。

4.3.1 比较价值迭代与策略迭代

首先，我们通过列出价值迭代和策略迭代算法的步骤来比较它们，如下所示。

- 策略迭代：选择任意初始策略 π_0 。在第 k 次迭代中，执行以下两个步骤。
  - 步骤 1：策略评估 (PE)。给定 π_k ，通过下式求解 v_{π_k}

$$v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

  - 步骤 2：策略改进 (PI)。给定 v_{π_k} ，通过下式求解 π_{k+1}

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k}).$$

- 价值迭代：选择任意初始价值 v_0 。在第 k 次迭代中，执行以下两个步骤。
  - 步骤 1：策略更新 (PU)。给定 v_k ，通过下式求解 π_{k+1}

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_k).$$

  - 步骤 2：价值更新 (VU)。给定 π_{k+1} ，通过下式求解 v_{k+1}

$$v_{k+1} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k.$$

上述两种算法的步骤可以图示如下：

$$\text{Policy iteration: } \pi_0 \xrightarrow{PE} v_{\pi_0} \xrightarrow{PI} \pi_1 \xrightarrow{PE} v_{\pi_1} \xrightarrow{PI} \pi_2 \xrightarrow{PE} v_{\pi_2} \xrightarrow{PI} \dots$$

$$\text{Value iteration: } v_0 \xrightarrow{PU} \pi_1' \xrightarrow{VU} v_1 \xrightarrow{PU} \pi_2' \xrightarrow{VU} v_2 \xrightarrow{PU} \dots$$

可以看出，这两种算法的过程非常相似。

我们更仔细地检查它们的价值步骤，以观察两种算法之间的差异。

特别地，让两种算法从*相同的初始条件*开始： $v_0 = v_{\pi_0}$ 。两种算法的过程列于表 4.6 中。在前三个步骤中，两种算法生成相同的结果，因为 $v_0 = v_{\pi_0}$ 。它们变得在第四步中有所不同。

	Policy iteration algorithm	Value iteration algorithm	Comments
1) Policy:	π_0	N/A	
2) Value:	$v_{\pi_0} = r_{\pi_0} + \gamma P_{\pi_0} v_{\pi_0}$	$v_0 \doteq v_{\pi_0}$	
3) Policy:	$\pi_1 = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_0})$	$\pi_1 = \arg\max_\pi (r_\pi + \gamma P_\pi v_0)$	The two policies are the same
4) Value:	$v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$	$v_1 = r_{\pi_1} + \gamma P_{\pi_1} v_0$	$v_{\pi_1} \geq v_1$ since $v_{\pi_1} \geq v_{\pi_0}$
5) Policy:	$\pi_2 = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_1})$	$\pi_2' = \arg\max_\pi (r_\pi + \gamma P_\pi v_1)$	
⋮	⋮	⋮	⋮

Table 4.6: A comparison between the implementation steps of policy iteration and value iteration.

表 4.6：策略迭代与价值迭代实现步骤的比较。

在第四步中，价值迭代算法执行 $v_1 = r_{\pi_1} + \gamma P_{\pi_1} v_0$ ，这是一个单步计算；

而策略迭代算法求解 $v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$ ，这需要无限次迭代。

如果我们显式写出第四步中求解 $v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$ 的迭代过程，一切就会变得清晰。令 $v_{\pi_1}^{(0)} = v_0$ ，我们有：

$$
\begin{aligned}
& & v_{\pi_1}^{(0)} &= v_0 \\
\text{价值迭代 (value iteration)} \leftarrow v_1 & \leftarrow & v_{\pi_1}^{(1)} &= r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(0)} \\
& & v_{\pi_1}^{(2)} &= r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(1)} \\
& & \vdots & \\
\text{截断策略迭代 (truncated policy iteration)} \leftarrow \bar{v}_1 & \leftarrow & v_{\pi_1}^{(j)} &= r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(j-1)} \\
& & \vdots & \\
\text{策略迭代 (policy iteration)} \leftarrow v_{\pi_1} & \leftarrow & v_{\pi_1}^{(\infty)} &= r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(\infty)}
\end{aligned}
$$

从上述过程中可以获得以下观察结果：

- 如果迭代*仅运行一次*，那么 $v_{\pi_1}^{(1)}$ 实际上就是 v_1 ，即价值迭代算法中计算的值。
- 如果迭代运行*无限次*，那么 $v_{\pi_1}^{(\infty)}$ 实际上就是 v_{π_1} ，即策略迭代算法中计算的值。
- 如果迭代运行*有限次数*（记为 $j_{truncate}$ ），那么这种算法称为*截断策略迭代 (truncated policy iteration)*。之所以称为*截断*，是因为从 $j_{truncate}$ 到 ∞ 的剩余迭代被截断了。

因此，价值迭代和策略迭代算法可以看作是截断策略迭代算法的两个极端情况：价值迭代在 $j_{\text{truncate}} = 1$ 处终止，而策略迭代在 $j_{\text{truncate}} = \infty$ 处终止。需要注意的是，尽管上述比较具有说明性，但它是基于 $v_{\pi_1}^{(0)} = v_0 = v_{\pi_0}$ 这一条件的。如果没有这个条件，这两个算法无法直接进行比较。

> **算法 4.3：截断策略迭代算法 (Truncated policy iteration algorithm)**

初始化：已知概率模型 $p(r|s,a)$ 和 $p(s'|s,a)$，对于所有 (s,a)。初始猜测 π_0。

目标：搜索最优状态价值和最优策略。

当 v_k 未收敛时，进行第 k 次迭代，做

策略评估 (Policy evaluation)：

初始化：选择初始猜测 $v_k^{(0)} = v_{k-1}$。最大迭代次数设定为 j_{truncate}。

当 $j < j_{\text{truncate}}$ 时，做

对于每一个状态 $s \in \mathcal{S}$，做

$$v_k^{(j+1)}(s) = \sum_a \pi_k(a|s) \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_k^{(j)}(s') \right]$$

设定 $v_k = v_k^{(j_{\text{truncate}})}$

策略改进 (Policy improvement)：

对于每一个状态 $s \in \mathcal{S}$，做

对于每一个动作 $a \in \mathcal{A}(s)$，做

$$q_k(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_k(s')$$

$$a_k^*(s) = \arg\max_a q_k(s,a)$$

如果 $a = a_k^*$，则 $\pi_{k+1}(a|s) = 1$，否则 $\pi_{k+1}(a|s) = 0$

4.3.2 截断策略迭代算法 (Truncated policy iteration algorithm)

简而言之，截断策略迭代算法与策略迭代算法相同，唯一的区别在于它在策略评估步骤中仅运行有限次数的迭代。其实现细节总结在算法 4.3 中。**值得注意的是，算法中的 v_k 和 $v_k^{(j)}$ 并不是状态价值。**相反，它们是真实状态价值的近似值，因为在策略评估步骤中仅执行了有限次数的迭代。

如果 v_k 不等于 v_{π_k}，算法是否仍然能够找到最优策略？答案是肯定的。直观地说，截断策略迭代介于价值迭代和策略迭代之间。一方面，它比价值迭代算法收敛得更快，因为它在策略评估步骤中计算了多次迭代。

另一方面，它比策略迭代算法收敛得更慢，因为它只计算有限次数的迭代。

图 4.5 说明了这种直觉。这种直觉也得到了以下分析的支持。

命题 4.1（价值改进 Value improvement）。考虑策略评估步骤：

$$v_{\pi_k}^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}, \quad j = 0, 1, 2, \ldots$$

如果初始猜测选择为 $v_{\pi_k}^{(0)} = v_{\pi_{k-1}}$，则对于 $j = 0, 1, 2, \ldots$，有

$$v_{\pi_k}^{(j+1)} \geq v_{\pi_k}^{(j)}$$

成立。

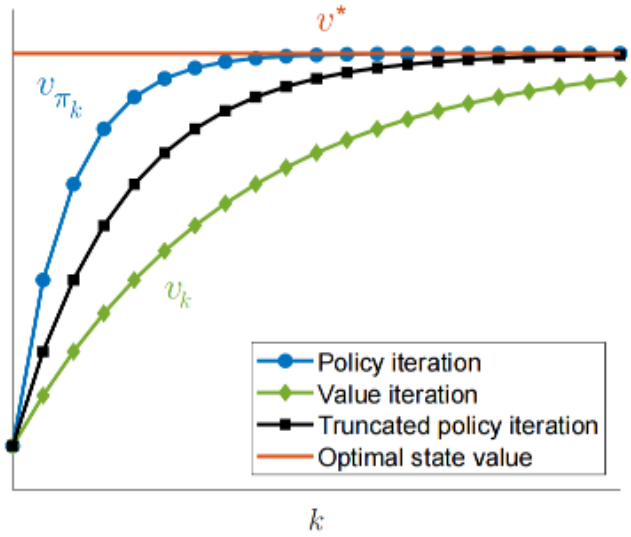

Figure 4.5: An illustration of the relationships between the value iteration, policy iteration, and truncated policy iteration algorithms.

图 4.5：价值迭代、策略迭代和截断策略迭代算法之间关系的图示。

> **方框 4.3：命题 4.1 的证明**
>
> 首先，由于 $v_{\pi_k}^{(j)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j-1)}$ 且 $v_{\pi_k}^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}$，我们有
>
> $$v_{\pi_k}^{(j+1)} - v_{\pi_k}^{(j)} = \gamma P_{\pi_k}(v_{\pi_k}^{(j)} - v_{\pi_k}^{(j-1)}) = \cdots = \gamma^j P_{\pi_k}^j (v_{\pi_k}^{(1)} - v_{\pi_k}^{(0)}). \quad (4.5)$$
>
> 其次，由于 $v_{\pi_k}^{(0)} = v_{\pi_{k-1}}$，我们有
>
> $$v_{\pi_k}^{(1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(0)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_{k-1}} \geq r_{\pi_{k-1}} + \gamma P_{\pi_{k-1}} v_{\pi_{k-1}} = v_{\pi_{k-1}} = v_{\pi_k}^{(0)},$$
>
> 其中不等式是由于 $\pi_k = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_{k-1}})$。将 $v_{\pi_k}^{(1)} \geq v_{\pi_k}^{(0)}$ 代入式 (4.5) 可得 $v_{\pi_k}^{(j+1)} \geq v_{\pi_k}^{(j)}$。

值得注意的是，命题 4.1 需要假设 $v_{\pi_k}^{(0)} = v_{\pi_{k-1}}$。然而，在实践中 $v_{\pi_{k-1}}$ 是无法获得的，只有 v_{k-1} 可用。尽管如此，**命题 4.1 仍然阐明了截断策略迭代算法的收敛性。**关于该主题更深入的讨论可以在 [2, Section 6.5] 中找到。

到目前为止，截断策略迭代的优势已经很明显了。相比于...

策略迭代算法相比，截断策略迭代算法在策略评估步骤中仅需要有限次数的迭代，因此计算效率更高。与价值迭代相比，截断策略迭代算法可以通过在策略评估步骤中多运行几次迭代来加快其收敛速度。

4.4 总结 (Summary)

本章介绍了三种可用于寻找最优策略的算法。

- 价值迭代 (Value iteration)：价值迭代算法与压缩映射定理建议的用于求解贝尔曼最优方程的算法相同。它可以分解为两个步骤：价值更新和策略更新。

- 策略迭代 (Policy iteration)：策略迭代算法比价值迭代算法稍微复杂一些。它也包含两个步骤：策略评估和策略改进。
- 截断策略迭代 (Truncated policy iteration)：价值迭代和策略迭代算法可以看作是截断策略迭代算法的两个极端情况。

这三种算法的一个共同特点是每次迭代都包含两个步骤。一步是更新价值，另一步是更新策略。价值更新和策略更新之间交互的思想广泛存在于强化学习算法中。这种思想也被称为*广义策略迭代 (generalized policy iteration)* [3]。

最后，本章介绍的算法需要系统模型。从第 5 章开始，我们将学习无模型 (model-free) 强化学习算法。我们将看到，可以通过扩展本章介绍的算法来获得无模型算法。

4.5 问与答 (Q&A)

- 问：价值迭代算法能保证找到最优策略吗？

答： 是的。这是因为价值迭代正是上一章中压缩映射定理建议的用于求解贝尔曼最优方程的算法。该算法的收敛性由压缩映射定理保证。

- 问：价值迭代算法生成的中间价值是状态价值吗？

答： 不是。这些价值不能保证满足任何策略的贝尔曼方程。

- 问：策略迭代算法包含哪些步骤？

答： 策略迭代算法的每次迭代包含两个步骤：策略评估和策略改进。在策略评估步骤中，算法旨在求解贝尔曼方程以获得当前策略的状态价值。在改进步骤中，算法旨在更新策略，以便新生成的策略具有更大的状态价值。

- 问：策略迭代算法中是否嵌入了另一个迭代算法？

答： 是的。在策略迭代算法的策略评估步骤中，需要一个迭代算法来求解当前策略的贝尔曼方程。

- 问：策略迭代算法生成的中间价值是状态价值吗？

答： 是的。这是因为这些价值是当前策略的贝尔曼方程的解。

- 问：策略迭代算法能保证找到最优策略吗？

答： 是的。我们在本章中给出了其收敛性的严格证明。

- 问：截断策略迭代算法和策略迭代算法之间有什么关系？

答： 顾名思义，截断策略迭代算法可以通过在策略评估步骤中仅执行有限次数的迭代，从策略迭代算法中获得。

- 问：截断策略迭代和价值迭代之间有什么关系？

答： 价值迭代可以看作是截断策略迭代的一种极端情况，即在策略评估步骤中仅运行一次迭代。

- 问：截断策略迭代算法生成的中间价值是状态价值吗？

答： 不是。只有当我们在策略评估步骤中运行无限次迭代时，才能获得真实的状态价值。如果我们运行有限次数的迭代，我们只能获得真实状态价值的近似值。

- 问：在截断策略迭代算法的策略评估步骤中，我们应该运行多少次迭代？

答： 一般的指导原则是运行几次迭代，但不要太多。在策略评估步骤中使用少量迭代可以加快整体收敛速度，但运行过多迭代并不会显著提高收敛速度。

- 问：什么是广义策略迭代 (generalized policy iteration)？

答： 广义策略迭代不是一个特定的算法。相反，它指的是价值更新和策略更新之间交互的一般思想。这个思想植根于策略迭代算法。本书介绍的大多数强化学习算法都属于广义策略迭代的范畴。

- **问：什么是基于模型的 (model-based) 和无模型的 (model-free) 强化学习？**

答： 虽然本章介绍的算法可以找到最优策略，但它们通常被称为动态规划算法，而不是强化学习算法，因为它们需要系统模型。强化学习算法可以分为两类：基于模型的 (model-based) 和无模型的 (model-free)。在这里，“基于模型”并不指的是需要系统模型。相反，基于模型的强化学习使用数据来估计系统模型，并在学习过程中使用该模型。相比之下，无模型强化学习在学习过程中不涉及模型估计。关于基于模型的强化学习的更多信息可以在 [13–16] 中找到。