第三章　动态规划寻找最优策略

本章将详细讲解如何利用动态规划算法来解决强化学习中的规划问题。“规划”是在已知环境动力学的基础上进行评估和控制，具体来说就是在了解包括状态和行为空间、转移概率矩阵、奖励等信息的基础上判断一个给定策略的价值函数，或判断一个策略的优劣并最终找到最优的策略和最优价值函数。尽管多数强化学习问题并不会给出具体的环境动力学，并且多数复杂的强化学习问题无法通过动态规划算法来快速求解，但本章在讲解利用动态规划算法进行规划的同时将重点阐述一些非常重要的概念，例如预测和控制、策略迭代、价值迭代等。正确理解这些概念对于了解本书后续章节的内容非常重要，因而可以说本章内容是整个强化学习核心内容的引子。

动态规划算法把求解复杂问题分解为求解子问题，通过求解子问题进而得到整个问题的解。在解决子问题的时候，其结果通常需要存储起来被用来解决后续复杂问题。当问题具有下列两个性质时，通常可以考虑使用动态规划来求解：第一个性质是一个复杂问题的最优解由数个小问题的最优解构成，可以通过寻找子问题的最优解来得到复杂问题的最优解；第二个性质是子问题在复杂问题内重复出现，使得子问题的解可以被存储起来重复利用。马尔科夫决策过程具有上述两个属性：贝尔曼方程把问题递归为求解子问题，价值函数相当于存储了一些子问题的解，可以复用。因此可以使用动态规划来求解马尔科夫决策过程。

预测和控制是规划的两个重要内容。预测是对给定策略的评估过程，控制是寻找一个最优策略的过程。对预测和控制的数学描述是这样：

预测 (prediction)：已知一个马尔科夫决策过程 MDP $\langle S, A, P, R, \gamma \rangle$ 和一个策略 π，或者是给定一个马尔科夫奖励过程 MRP $\langle S, P_\pi, R_\pi, \gamma \rangle$，求解基于该策略的价值函数 v_π。

控制 (control)：已知一个马尔科夫决策过程 MDP $\langle S, A, P, R, \gamma \rangle$，求解最优价值函数 v_* 和最优策略 π_*。

下文将详细讲解如何使用动态规划算法对一个 MDP 问题进行预测和控制。

3.1 策略评估

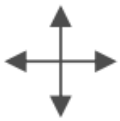
策略评估 (policy evaluation) 指计算给定策略下状态价值函数的过程。对策略评估，我们可以使用同步迭代联合动态规划的算法：从任意一个状态价值函数开始，依据给定的策略，结合贝尔曼期望方程、状态转移概率和奖励同步迭代更新状态价值函数，直至其收敛，得到该策略下最终的状态价值函数。理解该算法的关键在于在一个迭代周期内如何更新每一个状态的价值。该迭代法可以确保收敛形成一个稳定的价值函数，关于这一点的证明涉及到压缩映射理论，超出了本书的范围，有兴趣的读者可以查阅相关文献。

贝尔曼期望方程给出了如何根据状态转换关系中的后续状态 s' 来计算当前状态 s 的价值，在同步迭代法中，我们使用上一个迭代周期 k 内的后续状态价值来计算更新当前迭代周期 $k+1$ 内某状态 s 的价值：

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right) \tag{3.1}$$

我们可以对计算得到的新的状态价值函数再次进行迭代，直至状态函数收敛，也就是迭代计算得到每一个状态的新价值与原价值差别在一个很小的可接受范围内。

我们将用一个小型方格世界来解释同步迭代法进行策略评估的细节。在此之前，我们先详细描述下这个小型方格世界对应的强化学习问题，希望借此加深读者对于强化学习问题的理解。



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

图 3.1: 4×4 小型方格世界

考虑如图 3.1 所示的 4×4 的方格阵列，我们把它看成一个小世界。这个世界环境有 16 个状态，图中每一个小方格对应一个状态，依次用 0 – 15 标记它们。图中状态 0 和 15 分别位于左上角和右下角，是终止状态，用灰色表示。假设在这个小型方格世界中有一个可以进行上、下、左、右移动的个体，它需要通过移动自己来到达两个灰色格子中的任意一个来完成任务。这个小型格子世界作为环境有着自己的动力学特征：当个体采取的移动行为不会导致个体离开格子世界时，个体将以 100% 的几率到达它所要移动的方向的相邻的那个格子，之所以是相邻的格子而

不能跳格是由于环境约束个体每次只能移动一格，同时规定个体也不能斜向移动；如果个体采取会跳出格子世界的行为，那么环境将让个体以 100% 的概率停留在原来的状态；如果个体到达终止状态，任务结束，否则个体可以持续采取行为。每当个体采取了一个行为后，只要这个行为是个体在非终止状态时执行的，不管个体随后到达哪一个状态，环境都将给予个体值为 -1 的奖励值；而当个体处于终止位置时，任何行为将获得值为 0 的奖励并仍旧停留在终止位置。环境设置如此的奖励机制是利用了个体希望获得累计最大奖励的天性，而为了让个体在格子世界中用尽可能少的步数来到达终止状态，因为个体在世界中每多走一步，都将得到一个负值的奖励。为了简化问题，我们设置衰减因子 $\gamma = 1$。至此一个格子世界的强化学习问题就描述清楚了。

在这个小型格子世界的强化学习问题中，个体为了达到在完成任务时获得尽可能多的奖励（在此例中是为了尽可能减少负值奖励带来的惩罚）这个目标，它至少需要思考一个问题："当处在格子世界中的某一个状态时，我应该采取如何的行为才能尽快到达表示终止状态的格子。"这个问题对于拥有人类智慧的读者来说不是什么难题，因为我们知道整个世界环境的运行规律（动力学特征）。但对于格子世界中的个体来说就不那么简单了，因为个体身处格子世界中一开始并不清楚各个状态之间的位置关系，它不知道当自己处在状态 4 时只需要选择"向上"移动的行为就可以直接到达终止状态。此时个体能做的就是在任何一个状态时，它选择朝四个方向移动的概率相等。个体想到的这个办法就是一个基于**均一概率的随机策略**（uniform random policy）。个体遵循这个均一随机策略，不断产生行为，执行移动动作，从格子世界环境获得奖励（大多数是 -1 代表的惩罚），并到达一个新的或者曾经到达过的状态。长久下去，个体会发现：遵循这个均一随机策略时，每一个状态跟自己最后能够获得的最终奖励有一定的关系：在有些状态时自己最终获得的奖励并不那么少；而在其他一些状态时，自己获得的最终奖励就少得多了。个体最终发现，在这个均一随机策略指导下，每一个状态的价值是不一样的。这是一条非常重要的信息。对于个体来说，它需要通过不停的与环境交互，经历过多次的终止状态后才能对各个状态的价值有一定的认识。个体形成这个认识的过程就是策略评估的过程。而作为读者的我们，由于知晓描述整个格子世界的信息特征，不必要向格子世界中的个体那样通过与环境不停的交互来形成这种认识，我们可以直接通过迭代更新状态价值的办法来评估该策略下每一个状态的价值。

首先，我们假设所有除终止状态以外的 14 个状态的价值为 0。同时，由于终止状态获得的奖励为 0，我们可以认为两个终止状态的价值始终保持为 0。这样产生了第 $k = 0$ 次迭代的状态价值函数 (图 3.2(a))。

在随后的每一次迭代内，个体处于在任意状态都以均等的概率 (1/4) 选择朝上、下、左、右等四个方向中的一个进行移动；只要个体不处于终止状态，随后产生任意一个方向的移动后都将得到 -1 的奖励，并依据环境动力学将 100% 进入行为指向的相邻的格子或碰壁后留在原位，在更新某一状态的价值时需要分别计算 4 个行为带来的价值分量。在实践部分将详细演示价值迭代的计算过程。

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

(a) k=0

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

(b) k=1

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

(c) k=2

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

(d) k=3

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

(e) k=10

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

(f) k=∞

图 3.2: 小型方格世界迭代中的价值函数

3.2 策略迭代

完成对一个策略的评估，将得到基于该策略下每一个状态的价值。很明显，不同状态对应的价值一般也不同，那么个体是否可以根据得到的价值状态来调整自己的行动策略呢，例如考虑一种如下的贪婪策略：个体在某个状态下选择的行为是其能够到达后续所有可能的状态中价值最大的那个状态。我们以均一随机策略下第 2 次迭代后产生的价值函数为例说明这个贪婪策略。

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

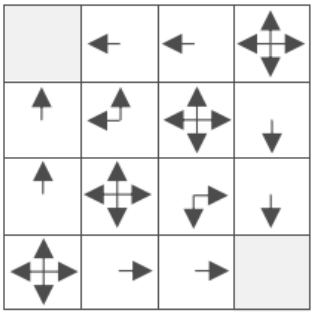


图 3.3: 小型方格世界策略的改善 (k=2)

如图 3.3 所示，右侧是根据左侧各状态的价值绘制的贪婪策略方案。个体处在任何一个状态时，将比较所有后续可能的状态的价值，从中选择一个最大价值的状态，再选择能到达这一状态的行为；如果有多个状态价值相同且均比其他可能的后续状态价值大，那么个体则从这多个最大

价值的状态中随机选择一个对应的行为。

在这个小型方格世界中，新的贪婪策略比之前的均一随机策略要优秀不少，至少在靠近终止状态的几个状态中，个体将有一个明确的行为，而不再是随机行为了。我们从均一随机策略下的价值函数中产生了新的更优秀的策略，这是一个策略改善的过程。

更一般的情况，当给定一个策略 π 时，可以得到基于该策略的价值函数 v_π，基于产生的价值函数可以得到一个贪婪策略 $\pi' = \text{greedy}(v_\pi)$。

依据新的策略 π' 会得到一个新的价值函数，并产生新的贪婪策略，如此重复循环迭代将最终得到最优价值函数 v^* 和最优策略 π^*。策略在循环迭代中得到更新改善的过程称为**策略迭代**（policy iteration）。图 3.4 直观地显示了策略迭代的过程。

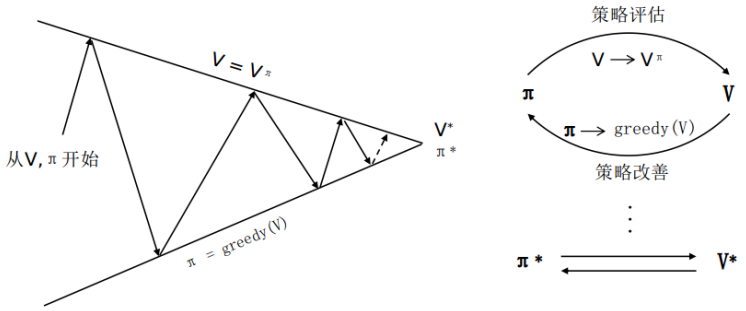


图 3.4: 策略迭代过程示意图

从一个初始策略 π 和初始价值函数 V 开始，基于该策略进行完整的价值评估过程得到一个新的价值函数，随后依据新的价值函数得到新的贪婪策略，随后计算新的贪婪策略下的价值函数，整个过程反复进行，在这个循环过程中策略和价值函数均得到迭代更新，并最终收敛值最有价值函数和最优策略。除初始策略外，迭代中的策略均是依据价值函数的贪婪策略。

下文给出基于贪婪策略的迭代将收敛于最优策略和最有状态价值函数的证明。

考虑一个依据确定性策略 π 对任意状态 s 产生的行为 $a = \pi(s)$，贪婪策略在同样的状态 s 下会得到新行为：$a' = \pi'(s)$，其中：

$$\pi'(s) = \underset{a \in A}{argmax}\ q_\pi(s, a) \tag{3.2}$$

假如个体在与环境交互的仅下一步采取该贪婪策略产生的行为，而在后续步骤仍采取基于**原**策略产生的行为，那么下面的（不）等式成立：

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

由于上式中的 s 对状态集 S 中的所有状态都成立，那么针对状态 s 的所有后续状态均使用贪婪策略产生的行为，不等式：$v_{\pi'} \geq v_{\pi}(s)$ 将成立。这表明新策略下状态价值函数总不劣于原策略下的状态价值函数。该步的推导如下：

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

如果在某一个迭代周期内，状态价值函数不再改善，即：

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

那么就满足了贝尔曼最优方程的描述：

$$v_{\pi} = \underset{a \in A}{max} q_{\pi}(s, a)$$

此时，对于所有状态集内的状态 $s \in S$，满足：$v_{\pi}(s) = v_{*}(s)$，这表明此时的策略 π 即为最优策略。证明完成。

3.3 价值迭代

细心的读者可能会发现，如果按照图 3.2 中第三次迭代得到的价值函数采用贪婪选择策略的话，该策略和最终的最优价值函数对应的贪婪选择策略是一样的，它们都对应于最优策略，如图 3.5，而通过基于均一随机策略的迭代法价值评估要经过数十次迭代才算收敛。这会引出一个问题：是否可以提前设置一个迭代终点来减少迭代次数而不影响得到最优策略呢？是否可以每迭代一次就进行一次策略评估呢？在回答这些问题之前，我们先从另一个角度剖析一下最优策略的意义。

任何一个最优策略可以分为两个阶段，首先该策略要能产生当前状态下的最优行为，其次对于该最优行为到达的后续状态时该策略仍然是一个最优策略。可以反过来理解这句话：如果一个策略不能在当前状态下产生一个最优行为，或者这个策略在针对当前状态的后续状态时不能产生一个最优行为，那么这个策略就不是最优策略。与价值函数对应起来，可以这样描述最优化原则：一个策略能够获得某状态 s 的最优价值当且仅当该策略也同时获得状态 s 所有可能的后续

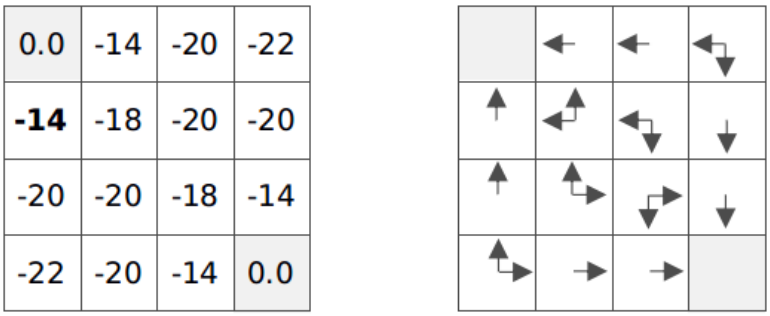


图 3.5: 小型方格世界 $k = \infty$ 时的贪婪策略

状态 s' 的最优价值。

对于状态价值的最优化原则告诉我们，一个状态的最优价值可以由其后续状态的最优价值通过前一章所述的贝尔曼最优方程来计算：

$$v_*(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \right)$$

这个公式带给我们的直觉是如果我们能知道最终状态的价值和相关奖励，可以直接计算得到最终状态的前一个所有可能状态的最优价值。更乐观的是，即使不知道最终状态是哪一个状态，也可以利用上述公式进行纯粹的价值迭代，不停的更新状态价值，最终得到最优价值，而且这种单纯价值迭代的方法甚至可以允许存在循环的状态转换乃至一些随机的状态转换过程。我们以一个更简单的方格世界来解释什么是单纯的价值迭代。

如图 3.6(V0) 所示是一个在 4×4 方格世界中寻找最短路径的问题。与本章前述的方格世界问题唯一的不同之处在于，该世界只在左上角有一个最终状态，个体在世界中需尽可能用最少步数到达左上角这个最终状态。

首先考虑到个体知道环境的动力学特征的情形。在这种情况下，个体可以直接计算得到与终止状态直接相邻（斜向不算）的左上角两个状态的最优价值均为 -1。随后个体又可以往右下角延伸计算得到与之前最优价值为 -1 的两个状态香相邻的 3 个状态的最优价值为 -2。以此类推，每一次迭代个体将从左上角朝着右下角方向依次直接计算得到一排斜向方格的最优价值，直至完成最右下角的一个方格最优价值的计算。

现在考虑更广泛适用的，个体不知道环境动力学特征的情形。在这种情况下，个体并不知道终止状态的位置，但是它依然能够直接进行价值迭代。与之前情形不同的是，此时的个体要针对所有的状态进行价值更新。为此，个体先随机地初始化所有状态价值 (V1)，示例中为了演示简便全部初始化为 0。在随后的一次迭代过程中，对于任何非终止状态，因为执行任何一个行为都将得到一个 -1 的奖励，而所有状态的价值都为 0，那么所有的非终止状态的价值经过计算后都

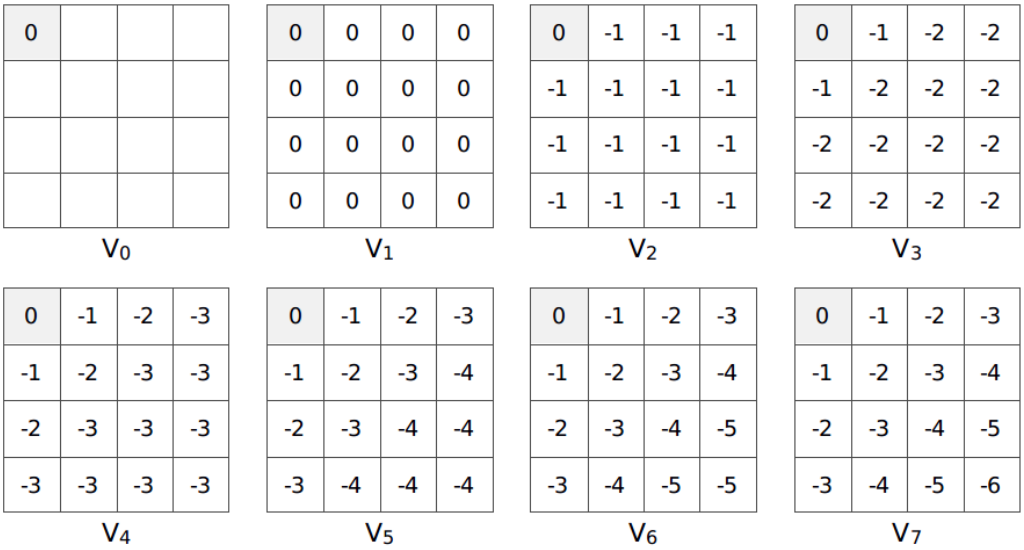


图 3.6: 小型方格世界最短路径问题

为 -1 (V2)。在下一次迭代中，除了与终止状态相邻的两个状态外的其余状态的价值都将因采取一个行为获得 -1 的奖励以及在前次迭代中得到的后续状态价值均为 -1 而将自身的价值更新为 -2；而与终止状态相邻的两个状态在更新价值时需将终止状态的价值 0 作为最高价值代入计算，因而这两个状态更新的价值仍然为 -1 (V3)。依次类推直到最右下角的状态更新为 -6 后 (V7)，再次迭代各状态的价值将不会发生变化，于是完成整个价值迭代的过程。

两种情形的相同点都是根据后续状态的价值，利用贝尔曼最优方程来更新得到前接状态的价值。两者的差别体现在：前者每次迭代仅计算相关的状态的价值，而且一次计算即得到最优状态价值，后者在每次迭代时要更新所有状态的价值。

可以看出价值迭代的目标仍然是寻找到一个最优策略，它通过贝尔曼最优方程从前次迭代的价值函数中计算得到当次迭代的价值函数，在这个反复迭代的过程中，并没有一个明确的策略参与，由于使用贝尔曼最优方程进行价值迭代时类似于贪婪地选择了最有行为对应的后续状态的价值，因而价值迭代其实等效于策略迭代中每迭代一次价值函数就更新一次策略的过程。需要注意的是，在纯粹的价值迭代寻找最优策略的过程中，迭代过程中产生的状态价值函数不一定对应一个策略。迭代过程中价值函数更新的公式为：

$$v_{k+1}(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right) \tag{3.3}$$

上述公式和图示中，k 表示迭代次数。

至此，使用同步动态规划进行规划基本就讲解完毕了。其中迭代法策略评估属于预测问题，

它使用贝尔曼期望方程来进行求解。策略迭代和价值迭代则属于控制问题，其中前者使用贝尔曼期望方程进行一定次数的价值迭代更新，随后在产生的价值函数基础上采取贪婪选择的策略改善方法形成新的策略，如此交替迭代不断的优化策略；价值迭代则不依赖任何策略，它使用贝尔曼最优方程直接对价值函数进行迭代更新。前文所述的这三类算法均是基于状态价值函数的，其每一次迭代的时间复杂度为 $O(mn^2)$，其中 m,n 分别为行为和状态空间的大小。读者也可以设计基于行为价值函数的上述算法，这种情况下每一次迭代的时间复杂度则变成了 $O(m^2n^2)$，本文不再详述。

3.4 异步动态规划算法

前文所述的系列算法均为同步动态规划算法，它表示所有的状态更新是同步的。与之对应的还有异步动态规划算法。在这些算法中，每一次迭代并不对所有状态的价值进行更新，而是依据一定的原则有选择性的更新部分状态的价值，这种算法能显著的节约计算资源，并且只要所有状态能够得到持续的被访问更新，那么也能确保算法收敛至最优解。比较常用的异步动态规划思想有：原位动态规划、优先级动态规划、和实时动态规划等。下文将简要叙述各类异步动态规划算法的特点。

原位动态规划 (in-place dynamic programming)：与同步动态规划算法通常对状态价值保留一个额外备份不同，原位动态规划则直接利用当前状态的后续状态的价值来更新当前状态的价值。

优先级动态规划 (prioritised sweeping)：该算法对每一个状态进行优先级分级，优先级越高的状态其状态价值优先得到更新。通常使用贝尔曼误差来评估状态的优先级，贝尔曼误差被表示为新状态价值与前次计算得到的状态价值差的绝对值。直观地说，如果一个状态价值在更新时变化特别大，那么该状态下次将得到较高的优先级再次更新。这种算法可以通过维护一个优先级队列来较轻松的实现。

实时动态规划 (real-time dynamic programming)：实时动态规划直接使用个体与环境交互产生的实际经历来更新状态价值，对于那些个体实际经历过的状态进行价值更新。这样个体经常访问过的状态将得到较高频次的价值更新，而与个体关系不密切、个体较少访问到的状态其价值得到更新的机会就较少。

动态规划算法使用**全宽度**（full-width）的回溯机制来进行状态价值的更新，也就是说，无论是同步还是异步动态规划，在每一次回溯更新某一个状态的价值时，都要追溯到该状态的所有可能的后续状态，并结合已知的马尔科夫决策过程定义的状态转换矩阵和奖励来更新该状态的价值。这种全宽度的价值更新方式对于状态数在百万级别及以下的中等规模的马尔科夫决策问题还是比较有效的，但是当问题规模继续变大时，动态规划算法将会因贝尔曼维度灾难而无法使

用，每一次的状态回溯更新都要消耗非常昂贵的计算资源。为此需要寻找其他有效的算法，这就是后文将要介绍的采样回溯。这类算法的一大特点是不需要知道马尔科夫决策过程的定义，也就是不需要了解状态转移概率矩阵以及奖励函数，而是使用采样产生的奖励和状态转移概率。这类算法通过采样避免了维度灾难，其回溯的计算时间消耗是常数级的。由于这类算法具有非常可观的优势，在解决大规模实际问题时得到了广泛的应用。

3.5 编程实践——动态规划求解小型方格世界最优策略

在本章的编程实践中，我们将结合 4*4 小型方格世界环境使用动态规划算法进行策略评估、策略迭代和价值迭代。本节将引导读者进一步熟悉马尔科夫决策过程的建模、熟悉动态规划算法的思想，巩固对贝尔曼期望方程、贝尔曼最优方程的认识，加深对均一随机策略、贪婪策略的理解。本节使用的代码与前节有许多相似的地方，但也有不少细微的差别。这些差别代表着我们逐渐从单纯的马尔科夫过程的建模逐渐转向强化学习的建模和实践中。

3.5.1 小型方格世界 MDP 建模

我们先对 4*4 小型方格世界的 MDP 进行建模，由于 4*4 方格世界环境简单，环境动力学明确，我们将不使用字典来保存状态价值、状态转移概率、奖励、策略等。我们使用列表来描述状态空间和行为空间，将编写一个反映环境动力学特征的方法来确定后续状态和奖励值，该方法接受当前状态和行为作为参数。状态转移概率和奖励将使用函数（方法）的形式来表达。代码如下：

```
S = [i for i in range(16)] # 状态空间
A = ["n", "e", "s", "w"] # 行为空间
# P,R,将由dynamics动态生成
ds_actions = {"n": -4, "e": 1, "s": 4, "w": -1} # 行为对状态的改变

def dynamics(s, a): # 环境动力学
    '''模拟小型方格世界的环境动力学特征
    Args:
        s 当前状态 int 0 - 15
        a 行为 str in ['n','e','s','w'] 分别表示北、东、南、西
    Returns: tuple (s_prime, reward, is_end)
        s_prime 后续状态
        reward 奖励值
        is_end 是否进入终止状态
```

```
    '''
    s_prime = s
    if (s%4 == 0 and a == "w") or (s<4 and a == "n") \
        or ((s+1)%4 == 0 and a == "e") or (s > 11 and a == "s")\
        or s in [0, 15]:
        pass
    else:
        ds = ds_actions[a]
        s_prime = s + ds
    reward = 0 if s in [0, 15] else -1
    is_end = True if s in [0, 15] else False
    return s_prime, reward, is_end

def P(s, a, s1): # 状态转移概率函数
    s_prime, _, _ = dynamics(s, a)
    return s1 == s_prime

def R(s, a): # 奖励函数
    _, r, _ = dynamics(s, a)
    return r

gamma = 1.00
MDP = S, A, R, P, gamma
```

最后建立的 MDP 同上一章一样是一个拥有五个元素的元组，只不过 R 和 P 都变成了函数而不是字典了。同样变成函数的还有策略。下面的代码分别建立了均一随机策略和贪婪策略，并给出了调用这两个策略的统一的接口。由于生成一个策略所需要的参数并不统一，例如像均一随机策略多数只需要知道行为空间就可以了，而贪婪策略则需要知道状态的价值。为了方便程序使用相同的代码调用不同的策略，我们对参数进行了统一。

```
def uniform_random_pi(MDP = None, V = None, s = None, a = None):
    _, A, _, _, _ = MDP
    n = len(A)
    return 0 if n == 0 else 1.0/n

def greedy_pi(MDP, V, s, a): # t贪婪策略
    S, A, P, R, gamma = MDP
    max_v, a_max_v = -float('inf'), []
```

```
    for a_opt in A:# 统计后续状态的最大价值以及到达到达该状态的行为（可能不止一个）
        s_prime, reward, _ = dynamics(s, a_opt)
        v_s_prime = get_value(V, s_prime)
        if v_s_prime > max_v:
            max_v = v_s_prime
            a_max_v = [a_opt]
        elif(v_s_prime == max_v):
            a_max_v.append(a_opt)
    n = len(a_max_v)
    if n == 0: return 0.0
    return 1.0/n if a in a_max_v else 0.0

def get_pi(Pi, s, a, MDP = None, V = None):
    return Pi(MDP, V, s, a)
```

在编写贪婪策略时，我们考虑了多个状态具有相同最大值的情况，此时贪婪策略将从这多个具有相同最大值的行为中随机选择一个。为了能使用前一章编写的一些方法，我们重写一下需要用到的诸如获取状态转移概率、奖励以及显示状态价值等的辅助方法：

```
# 辅助函数
def get_prob(P, s, a, s1): # 获取状态转移概率
    return P(s, a, s1)

def get_reward(R, s, a): # 获取奖励值
    return R(s, a)

def set_value(V, s, v): # 设置价值字典
    V[s] = v

def get_value(V, s): # 获取状态价值
    return V[s]

def display_V(V): # 显示状态价值
    for i in range(16):
        print('{0:>6.2f}'.format(V[i]),end = " ")
        if (i+1) % 4 == 0:
            print("")
```

```
    print()
```

有了这些基础，接下来就可以很轻松地完成迭代法策略评估、策略迭代和价值迭代了。在前一章的实践环节，我们已经实现了完成这三个功能的方法了，这里只要做少量针对性的修改就可以了，由于策略 Pi 现在不是查表式获取而是使用函数来定义的，因此我们需要做相应的修改，修改后的完整代码如下：

```
def compute_q(MDP, V, s, a):
    '''根据给定的MDP，价值函数V，计算状态行为对s,a的价值qsa
    '''
    S, A, R, P, gamma = MDP
    q_sa = 0
    for s_prime in S:
        q_sa += get_prob(P, s, a, s_prime) * get_value(V, s_prime)
        q_sa = get_reward(R, s,a) + gamma * q_sa
    return q_sa

def compute_v(MDP, V, Pi, s):
    '''给定MDP下依据某一策略Pi和当前状态价值函数V计算某状态s的价值
    '''
    S, A, R, P, gamma = MDP
    v_s = 0
    for a in A:
        v_s += get_pi(Pi, s, a, MDP, V) * compute_q(MDP, V, s, a)
    return v_s

def update_V(MDP, V, Pi):
    '''给定一个MDP和一个策略，更新该策略下的价值函数V
    '''
    S, _, _, _, _ = MDP
    V_prime = V.copy()
    for s in S:
        set_value(V_prime, s, compute_v(MDP, V_prime, Pi, s))
    return V_prime

def policy_evaluate(MDP, V, Pi, n):
    '''使用n次迭代计算来评估一个MDP在给定策略Pi下的状态价值，初始时价值为V
    '''
```

```
    for i in range(n):
        V = update_V(MDP, V, Pi)
    return V

def policy_iterate(MDP, V, Pi, n, m):
    for i in range(m):
        V = policy_evaluate(MDP, V, Pi, n)
        Pi = greedy_pi # 第一次迭代产生新的价值函数后随机使用贪婪策略
    return V

# 价值迭代得到最优状态价值过程
def compute_v_from_max_q(MDP, V, s):
    '''根据一个状态的下所有可能的行为价值中最大一个来确定当前状态价值
    '''
    S, A, R, P, gamma = MDP
    v_s = -float('inf')
    for a in A:
        qsa = compute_q(MDP, V, s, a)
        if qsa >= v_s:
            v_s = qsa
    return v_s

def update_V_without_pi(MDP, V):
    '''在不依赖策略的情况下直接通过后续状态的价值来更新状态价值
    '''
    S, _, _, _, _ = MDP
    V_prime = V.copy()
    for s in S:
        set_value(V_prime, s, compute_v_from_max_q(MDP, V_prime, s))
    return V_prime

def value_iterate(MDP, V, n):
    '''价值迭代
    '''
    for i in range(n):
        V = update_V_without_pi(MDP, V)
    return V
```

3.5.2 策略评估

接下来就可以来调用这些方法进行策略评估、策略迭代和价值迭代了。我们先来分别评估一下均一随机策略和贪婪策略下 16 个状态的最终价值:

```
V = [0  for _ in range(16)] # 状态价值
V_pi = policy_evaluate(MDP, V, uniform_random_pi, 100)
display_V(V_pi)

V = [0  for _ in range(16)] # 状态价值
V_pi = policy_evaluate(MDP, V, greedy_pi, 100)
display_V(V_pi)
# 将输出结果:
#   0.00 -14.00 -20.00 -22.00
# -14.00 -18.00 -20.00 -20.00
# -20.00 -20.00 -18.00 -14.00
# -22.00 -20.00 -14.00   0.00

#  0.00  -1.00  -2.00  -3.00
# -1.00  -2.00  -3.00  -2.00
# -2.00  -3.00  -2.00  -1.00
# -3.00  -2.00  -1.00   0.00
```

可以看出，均一随机策略下得到的结果与图 3.5 显示的结果相同。在使用贪婪策略时，各状态的最终价值与均一随机策略下的最终价值不同。这体现了状态的价值是基于特定策略的。

3.5.3 策略迭代

编写如下代码进行贪婪策略迭代，观察每迭代 1 次改善一次策略，共进行 100 次策略改善后的状态价值:

```
V = [0  for _ in range(16)] # 重置状态价值
V_pi = policy_iterate(MDP, V, greedy_pi, 1, 100)
display_V(V_pi)
# 将输出结果:
#  0.00  -1.00  -2.00  -3.00
# -1.00  -2.00  -3.00  -2.00
# -2.00  -3.00  -2.00  -1.00
```

```
# -3.00  -2.00  -1.00   0.00
```

3.5.4 价值迭代

下面的代码展示了单纯使用价值迭代的状态价值，我们把迭代次数选择为 4 次，可以发现仅 4 次迭代后，状态价值已经和最优状态价值一致了。

```
V_star = value_iterate(MDP, V, 4)
display_V(V_star)
# 将输出结果:
#  0.00  -1.00  -2.00  -3.00
# -1.00  -2.00  -3.00  -2.00
# -2.00  -3.00  -2.00  -1.00
# -3.00  -2.00  -1.00   0.00
```

我们还可以编写如下的代码来观察最优状态下对应的最优策略:

```
def greedy_policy(MDP, V, s):
    S, A, P, R, gamma = MDP
    max_v, a_max_v = -float('inf'), []
    for a_opt in A:# 统计后续状态的最大价值以及到达到达该状态的行为 (可能不止一个)
        s_prime, reward, _ = dynamics(s, a_opt)
        v_s_prime = get_value(V, s_prime)
        if v_s_prime > max_v:
            max_v = v_s_prime
            a_max_v = a_opt
        elif(v_s_prime == max_v):
            a_max_v += a_opt
    return str(a_max_v)

def display_policy(policy, MDP, V):
    S, A, P, R, gamma = MDP
    for i in range(16):
        print('{0:^6}'.format(policy(MDP, V, S[i])),end = " ")
        if (i+1) % 4 == 0:
```

```
            print("")
    print()

display_policy(greedy_policy, MDP, V_star)
# 将输出结果:
# nesw    w       w       sw
#  n      nw     nesw     s
#  n     nesw     es      s
#  ne     e       e      nesw
```

上面分别用 n,e,s,w 表示北、东、南、西四个行为。这与图 3.5 显示的结果是一致的。读者可以通过修改不同的参数或在迭代过程中输出价值和策略观察价值函数和策略函数的迭代过程。

Author:叶强qiangye@xmu.edu.cn

Author:叶强qiangye@gmail.com