Policy Evaluation

$$\pi \longrightarrow v_\pi$$

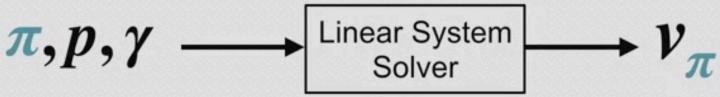
Recall that

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

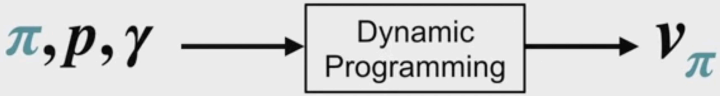
$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$ ⇒ sum of the future rewards

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

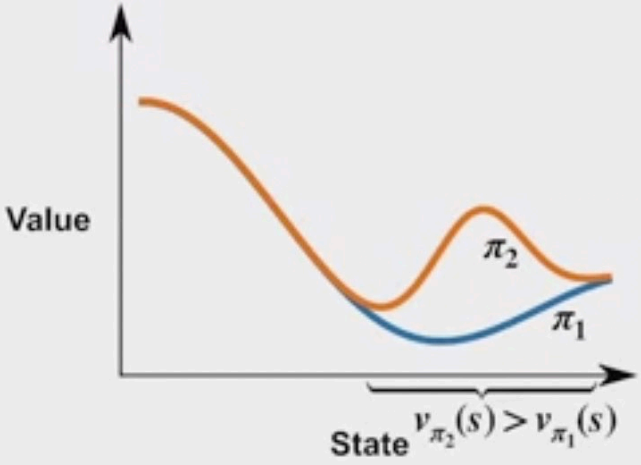
Recall that



In practice



Control is the task of improving a policy

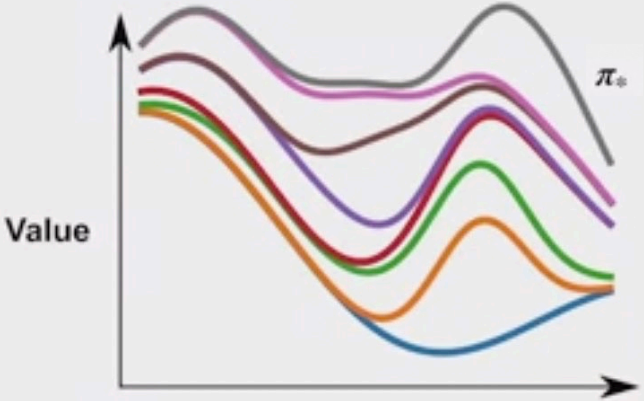


π_2 is strictly better than π_1

when π_2 is as good as or better than π_1, and there is a period that state value $V_{\pi_2}(s) > V_{\pi_1}(s)$.

Control is the task of improving a policy

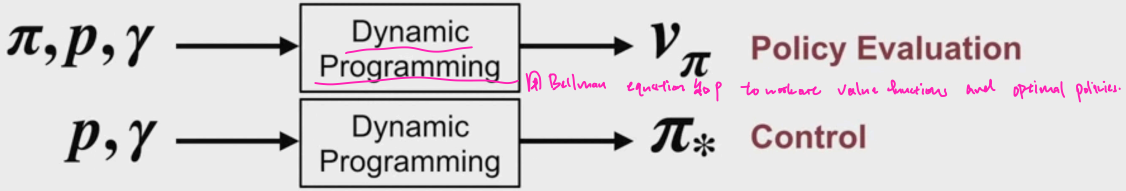
The goal of the control task is to modify a policy to produce a new one which is strictly better.



当 control is no longer possible 的时候, 也就是 there is no policy which is strictly better than the current policy.

∴ The current policy is optimal policy

∴ The control process complete.



用 Bellman equation 的 p to evaluate value functions and optimal policies.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

$$q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right]$$

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

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$$

Summary

- Policy evaluation is the task of determining the state-value function v_π, for a particular policy π
- Control is the task of improving an existing policy
- Dynamic programming techniques can be used to solve both these tasks, if we have access to the dynamics function p

动态编程算法(Dynamic programming algorithms) 是通过将贝尔曼方程转化为更新规则得到的。在本视频中，我们将介绍这些算法中的第一个，即迭代策略评估(iterative policy evaluation)

记住贝尔曼方程给了我们一个V_Pi的递归表达式。迭代策略评估的想法是如此简单，以至于一开始它可能看起来有点傻。我们采用贝尔曼方程，并直接将其作为更新规则。现在，我们有了一个程序，而不是一个对真实价值函数成立的方程，我们可以应用它来迭代完善我们对价值函数的估计。这将产生一连串越来越好的价值函数的近似值。让我们直观地看看这个程序是如何工作的。我们从我们的近似值函数的任意初始化开始，让我们称之为v_0。然后，每次迭代都通过使用幻灯片顶部所示的更新规则产生一个更好的近似值。每次迭代都会对状态空间中的每一个状态S进行更新，我们称之为扫荡(sweep)。反复应用这种更新会导致对状态值函数v_Pi的近似越来越好。如果这种更新使价值函数的近似值保持不变，也就是说，如果v_k加1等于所有状态的v_k，那么v_k等于v_Pi，我们就找到了价值函数。这是因为v_Pi是贝尔曼方程的唯一解。如果v_k已经服从贝尔曼方程，那么更新可能使v_k不发生变化的唯一方法就是v_k已经服从贝尔曼方程。事实上，可以证明，对于任何v_0的选择，当k接近无穷大时，v_k将在极限中收敛到v Pi。为了实现迭代策略评估，我们存储了两个数组，每个数组都有一个条目代表每个状态。一个数组，我们标记为V，存储当前的近似值函数。另一个数组，V prime，存储更新的值。通过使用两个数组，我们可以一次从旧的状态中计算出新的值，而在这个过程中旧的值不会被改变。

Iterative Policy Evaluation in a Nutshell

$$v_{\pi}(s) = \sum_{a} \pi(a \mid s) \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_{\pi}(s')\right]$$

↓

$$v_{k+1}(s) \leftarrow \sum_{a} \pi(a \mid s) \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_{k}(s')\right]$$

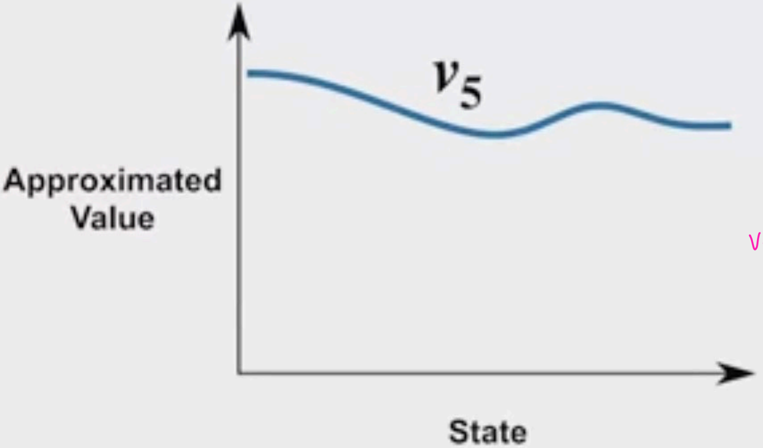
update rule，每 update 一次新的 state value 就是一个更好的近似值.

=> produce a sequence of better and better approximations to the value function.

a better approximation by using the update rule shown at the top of the slide. Each iteration applies this updates to every state, S, in the state space, which we call a sweep. Applying this update repeatedly leads to a better and better approximation to the state value function v Pi. If

$$v_{k+1}(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_k(s')\right]$$

=> 如果这个update function到后面停止变化了，即 $V_{k+1} = V_k$，所以我们就找到了 value function.

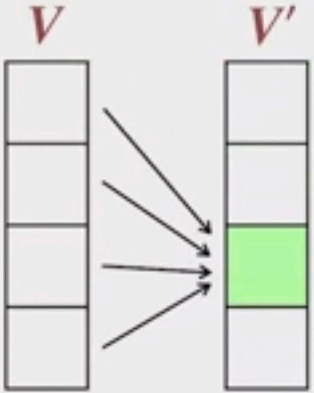


For any v_0

$$\lim_{k\to\infty} v_k = v_\pi$$

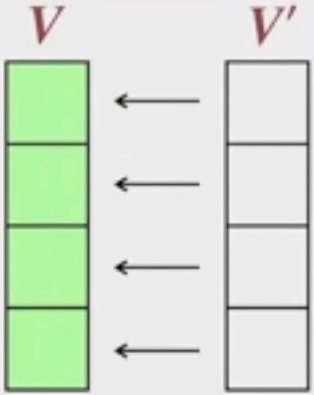
$V_k = V_\pi$ for k approaches infinity.

$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$



choice of v_0, v_k will converge to v Pi in the limit as k approaches infinity. To implement iterative policy evaluation, we store two arrays, each has one entry for every state. One array, which we label V stores the current approximate value function. Another array, V prime, stores the updated values. By using two arrays, we can compute the new values from the old one state at a time without the old values being changed in the process.

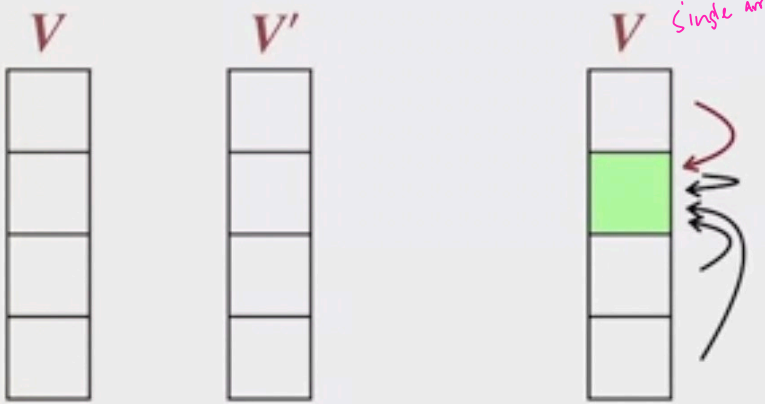
$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$



At the end of a full sweep, we can write all the new values into V; then we do the next iteration.

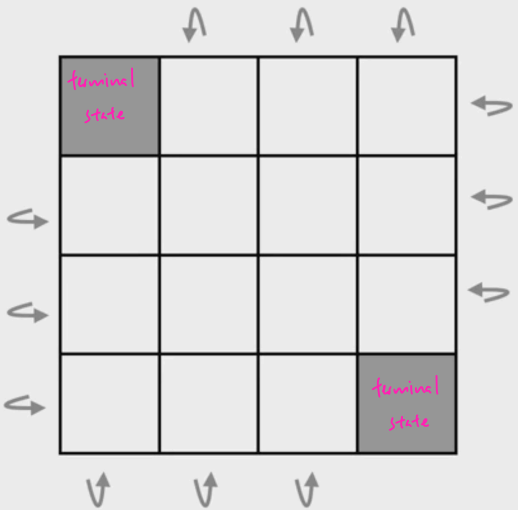
At the end of a full sweep, we can write all the new values into V; then we do the next iteration. It is also possible to implement a version with only one array, in which case, some updates will themselves use new values instead of old. This single array version is still guaranteed to converge, and in fact, will usually converge faster. This is because it gets to use the updated values sooner. But for simplicity, we focus on the two array version. Let's look at how iterative

$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$



在一个完整的扫描结束后，我们可以把所有的新值写进V中；然后进行下一次迭代。我们也可以实现一个只有一个数组的版本，在这种情况下，一些更新本身就会使用新值而不是旧值。这种单数组版本仍然可以保证收敛，而且事实上，通常会收敛得更快。这是因为它能更快地使用更新的值。但是为了简单起见，我们把重点放在双数组版本上。

让我们看看迭代策略评估在一个特定的例子中是如何工作的。考虑一下这里显示的四乘四的网格世界。这是一个偶发的MDP，终端状态位于左上角和右下角。终端状态被显示在两个地方，但从形式上看是同一个状态。每一次转换的奖励都是减一。由于这个问题是偶发性的，让我们考虑gamma等于1的未贴现情况。在每个状态下有四个可能的行动，即上、下、左、右。每个行动都是确定性的。如果该行动会使代理人离开网格，它反而会使代理人处于相同的状态。现在，让我们评估一下均匀随机策略(uniform random policy)，它在四分之一的时间内选择四个行动中的每一个。值函数表示从一个给定的状态到终止的预期步骤数。我们扫过状态的顺序并不重要，因为我们使用的是算法的两个阵列版本。让我们假设我们先从左到右扫过这些状态，然后再从上到下。我们从不更新终端状态的值，因为它被定义为零。我们将V中的所有数值初始化为零。存储在 V' 中的初始值是相关的，因为它们在被使用之前总会被更新。现在我们可以开始我们的第一次迭代，对状态一进行更新。为了计算更新，我们必须对所有行动进行求和。首先考虑左边的行动，在统一的随机策略下，它的概率为四分之一。动态函数P在这里是确定的，所以只有奖励和1S素数的值对总和有贡献。总和包括奖励的减一，和终端状态的值的零。由于我们把所有的状态值都初始化为零，而且每次转换的奖励都是减一，所以所有其他行动的计算结果都是一样的。结果是，状态一的V'被设置为负一。



$R = -1$ ⇒ 매 action마다 -1의 reward를 받게되는 것을 의미한다.

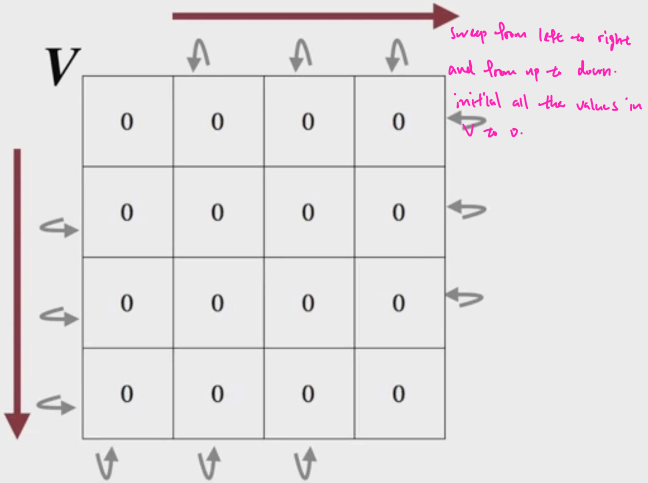
$\gamma = 1$

Policy

uniform random policy.

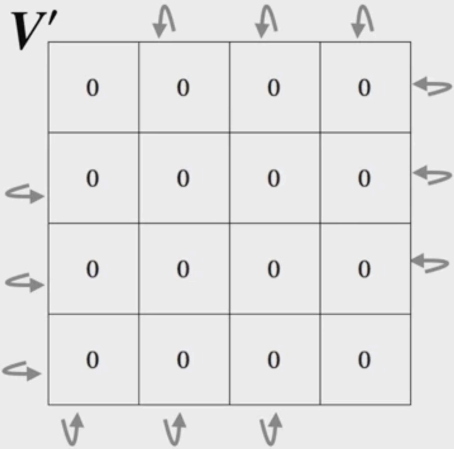
$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$

2) use the two array algorithm to update.



sweep from left to right and from up to down.

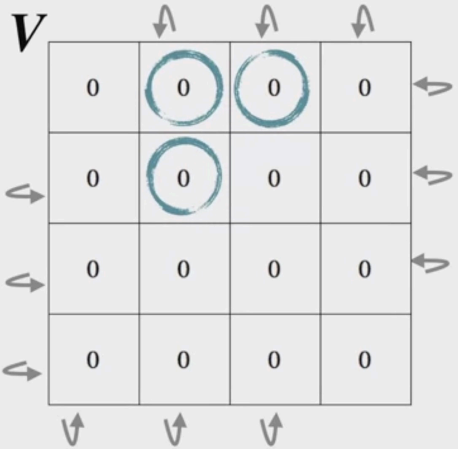
initial all the values in V to 0.

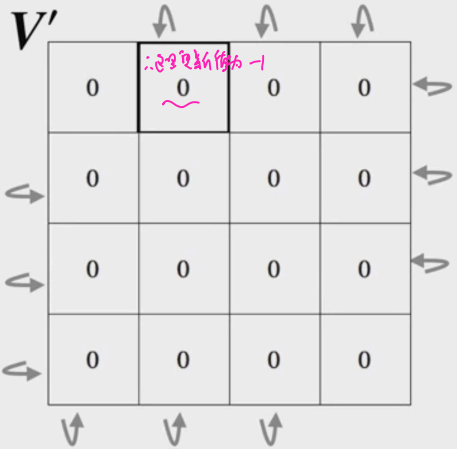


$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$

$$0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) = -1$$

reward

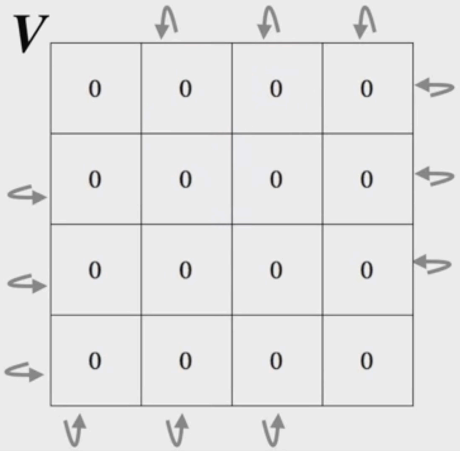


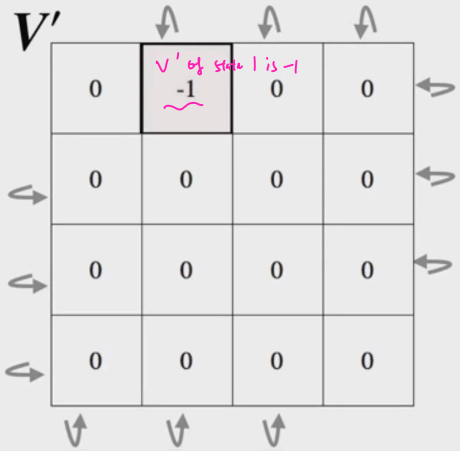


5:16 / 8:39

$$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma V(s')\right]$$

$$0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) = -1$$





$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$

$0.25 * (-1 + 0)$

Value of v_1 in the update is still 0 because it's using the previous value.

V

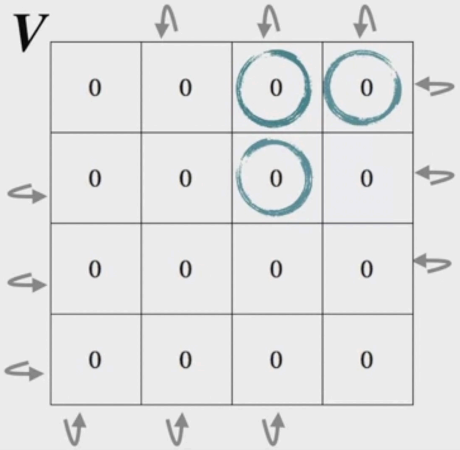
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

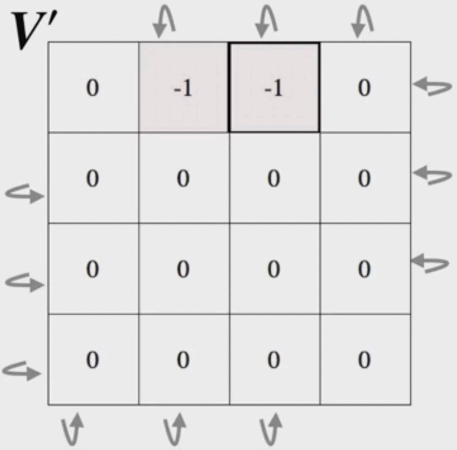
V'

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

$$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma V(s')\right]$$

$$0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) = -1$$





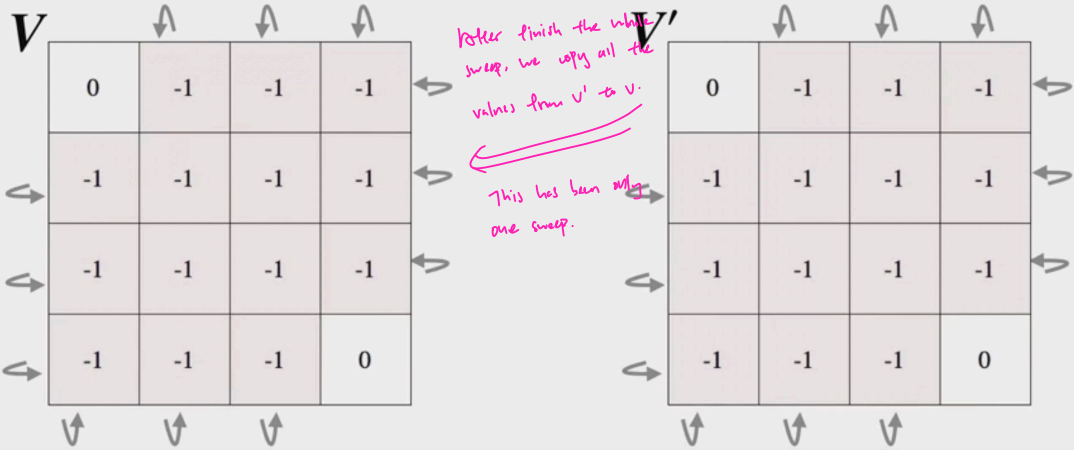
$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$

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

v_1: 经过sweep 1

v_2: 经过sweep 2

v_π: 最终得到的V_π.



Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated ⇒ take any policy we want to evaluate

$V \leftarrow \vec{0}, V' \leftarrow \vec{0}$ ⇒ initialize V and V' (初值是0).

Loop: ⇒ continue until the change of the update function V' becomes small.

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

$$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$$

$\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$ ⇒ track the largest update to this state value in a given iteration.

$V \leftarrow V'$

until $\Delta < \theta$ (a small positive number) — the stopping parameter, 如 $\theta = 0.001$ (θ越小，最终得到的结果越准确).

Output $V \approx v_\pi$

The outer loop terminates when this maximum change is less than some user-specified constant called theta. As discussed before, once the approximate value function stops changing, we have converged to V Pi. Similarly, once the change in the approximate value function is very small, this means we are close to V Pi. Let's

6:24 / 8:39

接下来，我们转到状态二。我们首先评估左边行动的和中的项。同样，行动的概率是四分之一，在这种情况下，下一个状态是状态一。虽然我们已经更新了状态一的值，但我们正在运行的算法版本将使用存储在V中的旧值，所以在更新中状态一的值仍然为零。同样，所有其他的动作看起来也是一样的。其结果是，状态二的V素也被设置为负一。事实上，由于每个状态的值都被初始化为零，每个状态的值都将被设置为负一。在完成这个完整的扫描后，我们将更新的值从V'复制到V。现在我们来讨论迭代策略评估的完整算法。采取任何我们想要评估的策略，初始化两个数组V和V prime。我们可以随心所欲地初始化它们，但让我们把它们设置为零。我们刚刚看到了迭代策略评估的一个步骤是如何进行的。让我们来看看我们如何计算多次扫描，并确定算法如何停止。外循环一直持续到近似值函数的变化变得很小。我们跟踪该状态值在特定迭代中的最大更新。我们把这个称为delta。当这个最大变化小于用户指定的常数theta时，外循环终止。

如前所述，一旦近似值函数停止变化，我们就已经收敛到了V Pi。同样地，一旦近似值函数的变化非常小，这意味着我们已经接近V Pi。让我们继续我们在网格世界的例子中的话题。我们刚刚完成了我们的第一次清扫。让我们用我们的停顿参数theta的值0.001来表示。我们选择的数值越小，我们的最终数值估计就越准确。我们已经完成了一次迭代，数值的最大变化是1.0。由于这大于0.001，我们继续进行下一次迭代。在第二次扫频之后，注意终端状态是如何首先开始影响最近的状态的值的。让我们再进行一次扫频。我们看到，现在终端状态的影响已经进一步扩散了。让我们再运行几次扫频，看看会发生什么。我们可以开始看到每个状态的值是如何与它与终端状态的接近程度相关的。让我们继续运行，直到我们的最大delta小于theta。这是我们最终得出的结果，我们的近似值函数(approximate value function)已经收敛到了随机策略的值函数(value function for the random policy)，我们完成了。

$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$

$\theta = 0.001$ $\Delta = 1.0$ to sweep 1次, largest change in update = 1. 1 > 0.001, ∴ 继续循环.

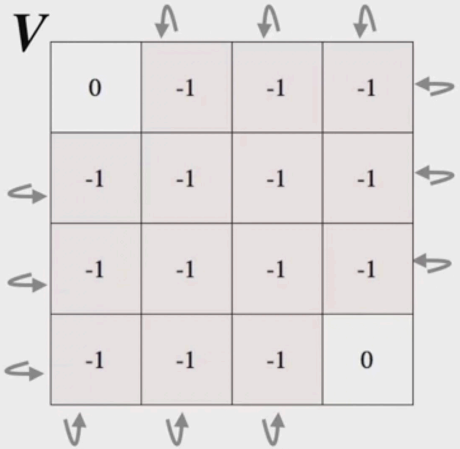
V

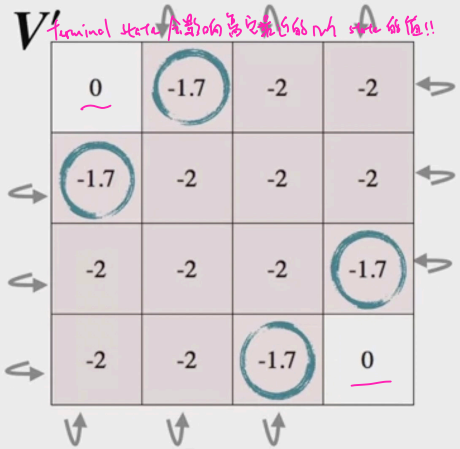
0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0

V'

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

$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma V(s')\right]$$





$$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma V(s')\right]$$

$$\Delta < 0.001$$

V

0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0

V'

0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0

Summary

- We can turn the Bellman equation into an update rule, to iteratively compute value functions

我们刚刚看了动态编程如何被用来迭代评估一个政策。我们暗示，这是实现控制任务的第一步，或者说目标是改进一个政策。在这段视频中，我们将最终解释这是如何工作的。

之前，我们表明，给定v*，我们可以通过选择贪婪行动找到最佳政策。贪婪行动在每个状态下都能使贝尔曼最优方程最大化。想象一下，我们不选择最优价值函数，而是选择一个相对于任意政策Pi的价值函数v Pi来说是贪婪的行动。对于这个新政策，我们可以说什么呢？它对v_Pi来说是贪婪的。首先要注意的是，这个新政策必须与Pi不同。如果这个贪婪化并没有改变Pi，那么Pi对于它自己的价值函数来说已经是贪婪的。这只是说v_Pi服从贝尔曼最优方程的另一种方式。在这种情况下，Pi已经是最优的。事实上，以这种方式得到的新政策必须是对Pi的严格改进，除非Pi已经是最优的。这是一个叫做政策改进定理的一般结果的结果。

Recall that Greedy action ⇒ maximize the Bellman's optimality equation in each state.

$$\pi_*(s) = \operatorname*{argmax}_a \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_*(s')\right]$$

?

$$\operatorname*{argmax}_a \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]$$

Imagine instead of the optimal value function, we select an action which is greedy with respect to the value function v Pi of an arbitrary policy Pi. What can we say about this new policy? That it is greedy with respect to v Pi. The first thing to note is that this new policy must be different than Pi. If this greedification doesn't change Pi, then Pi was already greedy with respect to its own value function. This is just another way of saying that v Pi obeys the Bellman's optimality equation. In

↳ 21 π is already optimum.

$$\pi(s) = \operatorname*{argmax}_a \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \text{ for all } s \in \mathcal{S}$$

→ v_π obeys the Bellman optimality equation

回顾一下q_Pi的定义。它告诉你，如果你采取行动A，然后遵循政策Pi时，一个状态的价值。想象一下，我们根据Pi prime采取行动A，然后遵循政策Pi。如果这个行动比Pi下的行动有更高的价值，那么Pi prime肯定更好。政策改进定理正式说明了这个想法。如果在每个状态下，Pi prime选择的行动的价值大于或等于Pi选择的行动的价值，那么政策Pi prime至少和Pi一样好。如果价值严格大于和至少一个状态，那么政策Pi prime严格来说是更好的。让我们看看这在我们之前使用的四乘四网格卷上是如何运作的。这是我们找到的最终价值函数。记住，这是统一随机策略的价值函数。现在，贪婪的Pi策略可能是什么样子的？在每个状态下，我们需要选择导致下一个状态的最高值的行动。在这种情况下，就是负值最小的那个值。这里是Pi'。这与我们开始时的均匀随机策略完全不同。要知道，这里显示的值与Pi'的值不一致。根据策略改进定理，新的策略保证是对我们开始使用的统一随机策略的改进。事实上，如果你更仔细地看一下新政策，我们可以看到它实际上是最优的。在每个状态下，所选择的行动都位于通往终端状态的最短路径上。请记住，我们开始时的价值函数并不是最优的价值函数，然而，关于v_Pi的贪婪策略是最优的。更一般地说，政策改进定理只保证新政策是对原来政策的改进。我们不能总是期望那么容易地找到最优政策。这段视频就到此为止。

你现在应该明白，政策改进定理(policy improvement theorem)告诉我们，greedified pi政策是一个严格的改进，除非原始政策已经是最优的。

must be

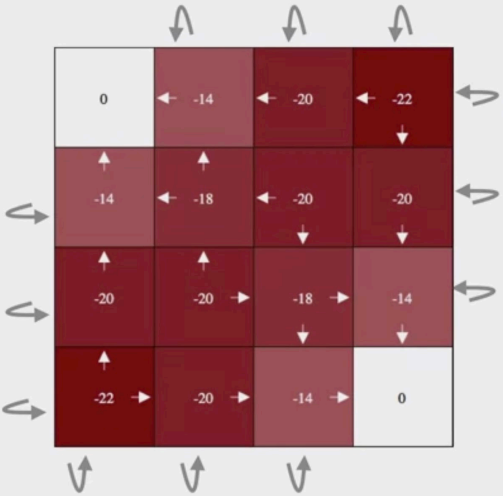
The new policy is a strict improvement over π unless π is already optimal

Policy Improvement Theorem

$$q_\pi(s, \pi'(s)) \geq q_\pi(s, \pi(s)) \text{ for all } s \in \mathcal{S} \rightarrow \pi' \geq \pi$$

$$q_\pi(s, \pi'(s)) > q_\pi(s, \pi(s)) \text{ for at least one } s \in \mathcal{S} \rightarrow \pi' > \pi$$

prime must be better. The policy improvement theorem formalizes this idea. Policy Pi prime is at least as good as Pi if in each state, the value of the action selected by Pi prime is greater than or equal to the value of the action selected by Pi. Policy pi prime is strictly better if the value is strictly greater and at least one state. Let's see how this works on the four-by-four grid rolled we



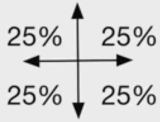
for all $s \in \mathcal{S}$

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_{\pi}(s')\right]$$

$\pi' > \pi$

$R = -1$

π



Summary

- The Policy Improvement Theorem tells us that a greedified policy is a strict improvement
- Use the value function under a given policy, to produce a strictly better policy

我们刚刚学习了如何利用为给定策略计算的价值函数来寻找更好的策略。在这段视频中，我们将展示如何利用这一点，通过迭代评估和证明一连串的政策来找到最佳政策。

回顾一下政策改进定理(policy improvement theorem)。它告诉我们，我们可以通过对给定政策的价值函数的贪婪行为来构建一个严格意义上更好的政策，除非给定政策已经是最优的。假设我们从政策Pi 1开始。我们可以使用迭代政策评估Pi 1，以获得状态值V_Pi 1。 我们称这为评估步骤。利用政策改进定理的结果，我们可以对V_Pi 1进行贪婪的评估，以获得更好的政策，Pi 2。我们称此为改进步骤(the improvement step)。然后我们可以计算V_Pi 2，并使用它来获得一个更好的政策，Pi 3。这就给了我们一连串的更好的政策。每个政策都保证是对上一个政策的改进，除非上一个政策已经是最优的。因此，当我们完成一个迭代，并且政策保持不变，我们知道我们已经找到了最佳政策。在这一点上，我们可以终止该算法。以这种方式生成的每个策略都是确定性的。确定性政策的数量是有限的，所以这种迭代改进最终必须达到一个最优政策。这种寻找最优策略的方法被称为策略迭代(policy iteration)。

Policy Iteration

Evaluation

Improvement

evaluation step

(用 iteration policy 评估 π_1，以获得 state value V_{π_1})

$$
\begin{array}{ccccccccccccccc}
 & E & & I & & E & & I & & E & & I & & E & & I & \\
\pi_1 & \rightarrow & v_{\pi_1} & \rightarrow & \pi_2 & \rightarrow & v_{\pi_2} & \rightarrow & \pi_3 & \rightarrow & \dots & \rightarrow & \pi_* & \rightarrow & v_{\pi_*} & \rightarrow & \pi_*
\end{array}
$$

improvement step

(将 V_{π_1} 进行 greedy 的评估，以获得更好的 policy π_2).

此时，π_2 对于 π_1 的 value function 来说是 greedy 的。但 V_{π_1} 不再准确反映 π_2 的价值.

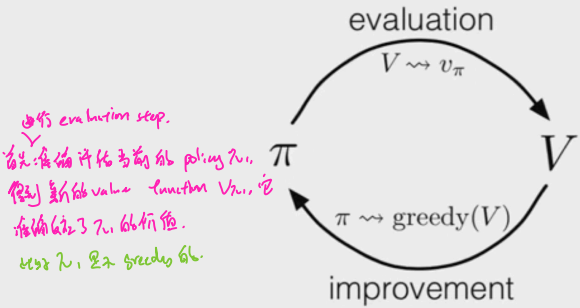
Deterministic

用 policy iteration 找出来的所有 policy 都是确定性的.

政策迭代由两个不同的步骤组成，即评估(evaluation)和改进(improvement)，反复进行。我们首先评估我们当前的政策，Pi 1，这给了我们一个新的价值函数，准确地反映了Pi 1的价值。然后改进步骤使用V Pi 1来产生一个贪婪的政策Pi 2。此时，相对于Pi 1的价值函数，Pi 2是贪婪的，但V Pi 1不再准确反映Pi 2的价值。下一步的评估使我们的价值函数相对于政策Pi 2来说是准确的。一旦我们这样做，我们的政策就再次不贪婪了。这种政策和价值的舞蹈来回进行，直到我们达到唯一的政策，即对它自己的价值函数而言是贪婪的，即最优政策(optimal policy)。在这一点上，也只有在这一点上，政策是贪婪的，价值函数是准确的。我们可以把这种舞蹈想象成在价值函数准确的一条线和政策贪婪的另一条线之间来回跳动。这两条线只在最佳政策和价值函数处相交。政策迭代总是通过首先投射到v等于v Pi的那条线上，然后投射到Pi对v来说是贪婪的那条线上，从而向交点推进。下面是这个程序的伪代码。

Policy Iteration

$$\pi_1 \longrightarrow v_{\pi_1}$$



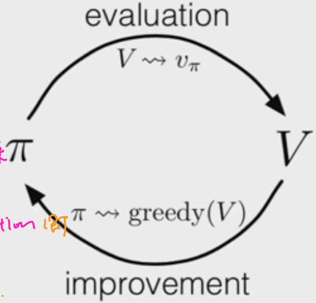
Policy Iteration

$$\pi_2 \longleftarrow v_{\pi_1}$$

⇒ 然后 improvement step 用 V_{π_1} 来
算一个新的 greedy policy π_2.

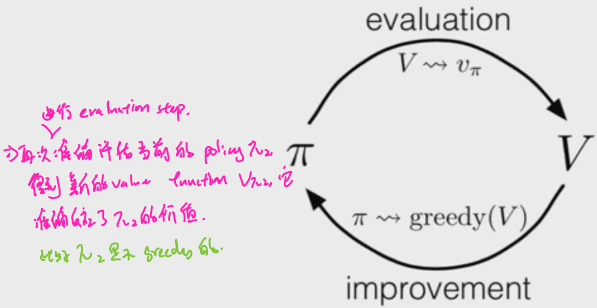
比如 π_2 对于 π_1 的 value function (即 V_{π_1}) 来说是 greedy 的.

但 V_{π_1} 不再准确反映 π_2 的价值.

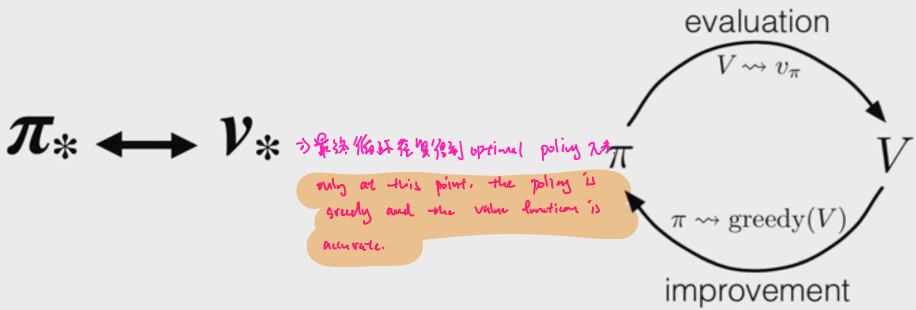


Policy Iteration

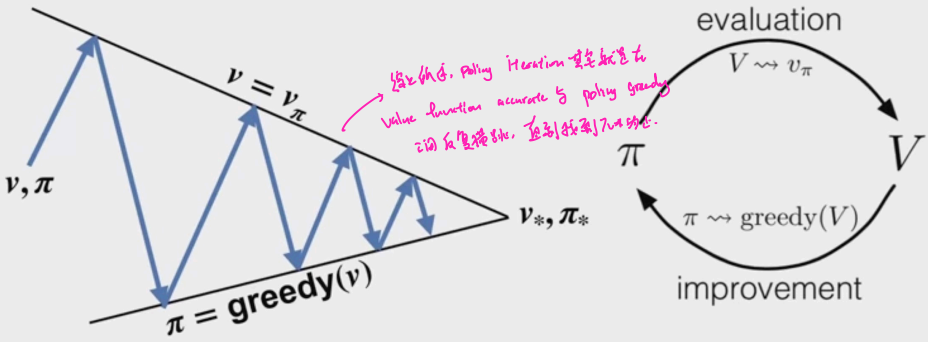
$\pi_2 \longrightarrow v_{\pi_2}$



Policy Iteration



Policy Iteration



3:11 / 8:22

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

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

   We initialize v and Pi in any way we like for each state s.

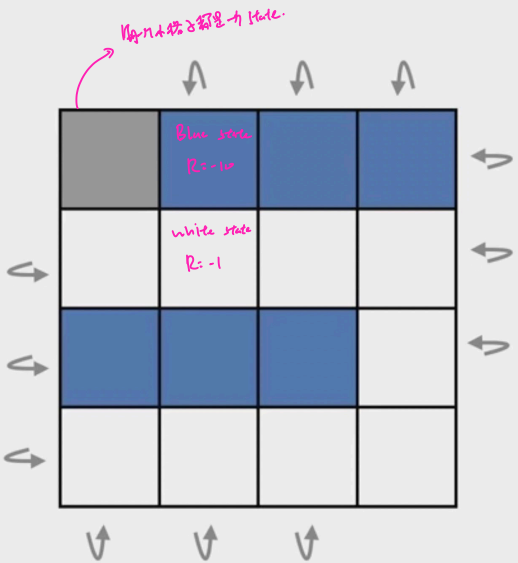
2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

   Next, we call iterative policy evaluation to make V reflect the value of Pi.

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

   Then, in each state, we set Pi to select the maximizing action under the value function. If this procedure changes the selected action in any state, we note that the policy is still changing, and set policy stable to force. After completing step 3, we check if the policy is stable. If not, we carry on and evaluate the new policy.

我们以任何方式为每个状态s初始化v和Pi。接下来，我们调用迭代策略评估，使V反映Pi的值。这就是我们在本模块中早先学到的算法。然后，在每个状态下，我们设置Pi来选择价值函数下的最大化行动。如果这个程序在任何状态下改变了所选择的行动，我们注意到政策仍在变化，并将政策稳定设置为强制。完成第三步后，我们检查政策是否稳定。如果不是，我们继续进行并评估新的策略。让我们看看这在一个简单问题上是如何工作的，以建立一些直觉。还记得我们用来演示迭代策略评估的四乘四网格规则的例子吗？之前，我们表明，通过评估随机策略，并只贪婪一次，我们可以找到最优策略。对于策略迭代来说，这并不是一个非常有趣的案例。让我们稍微修改一下这个问题，使控制任务更难一些。首先，让我们去掉其中一个终端状态，这样就只有一种方式来结束这一集。在此之前，每个状态都有一个减1的奖励。相反，让我们增加一些特别糟糕的状态。这些坏状态用蓝色标记。过渡到这些状态会得到负10的奖励。最佳策略应该沿着白色的蜿蜒的低成本路径到达终端状态。这种额外的复杂性意味着策略的迭代需要多次迭代才能发现路径。

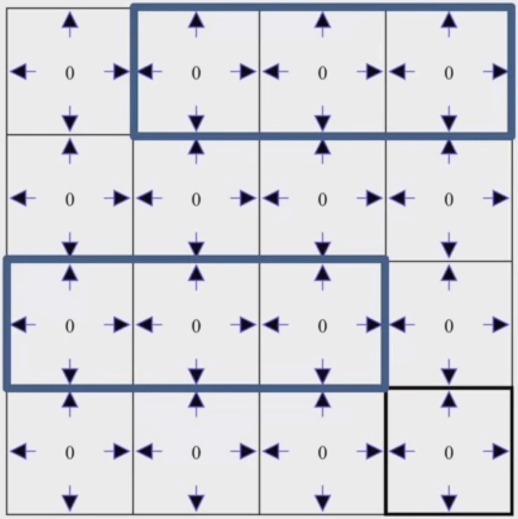


$$R = \begin{cases} -10 & \text{Blue states} \\ -1 & \text{Other states} \end{cases}$$

$$\gamma = 1$$

让我们看看这一点是如何实现的。首先，我们初始化一个策略和价值函数。像以前一样，我们选择统一的随机策略(uniform random policy)，并将所有状态下的价值估计设置为零。第一步是使用迭代策略评估(iterative policy evaluation)来评估统一随机策略。既然你已经看到了这是如何工作的，让我们直接跳到结果。这些值在任何地方都是相当负的，在接近目标的状态下，这些值略低。接下来我们进行改进步骤。你之前已经看到了greedification是如何工作的。所以，我们再次跳到结果。请注意，在终端状态附近，策略正确地遵循了通往终端状态的低费用路径。然而，在最下面一行的状态中，该策略却采取了更直接的、但价值更低的路径穿过坏状态。让我们评估一下这个新策略。注意到在经过一次改进后，数值开始变得更加合理，但我们还没有完成。让我们再贪婪一次。记住，政策改进定理告诉我们，这个新政策是对上一个政策的改进，除非我们已经达到了最优。具体来说，右下角的状态现在是沿着低成本的路径直线上升。政策评估的另一个步骤反映了这种变化。右下角状态的值从负15变成了仅仅负6。让我们继续下去，直到我们达到最佳政策。再改进一步，改善行动选择，又是一个状态。下一步的政策评估反映了这种变化，再次改进，评估，再改进。现在，我们可以看到，政策已经达到了最佳状态，并且遵循一条避免蓝色状态的低成本路径。再评估一次，我们就得到了最优的价值函数。如果我们再次尝试贪婪化，政策仍然没有改变。这告诉我们，策略迭代已经完成，最优策略已经被找到。这个例子显示了策略迭代的力量，因为它保证我们可以遵循一连串越来越好的策略，直到我们达到一个最优策略。

政策迭代切入了搜索空间(search space)，这在最优政策不直接的情况下是很关键的，在这个例子中就是如此。同样的复杂性会出现，我们真正关心的问题也会出现。你现在应该明白政策迭代是如何在政策评估和政策改进之间交替进行的，以及政策迭代如何遵循更好的政策和价值函数序列，直到达到最佳政策和最佳价值函数。



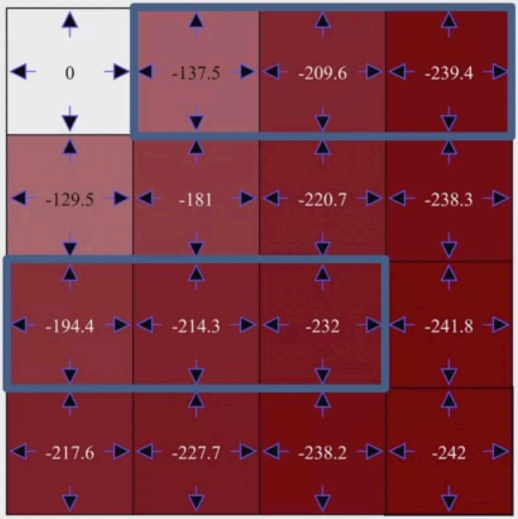
initialize all state values to 0.

Policy Iteration

Evaluation

Improvement

⇒ Guaranteed to find the optimal policy in time polynomial in the number of states and actions.

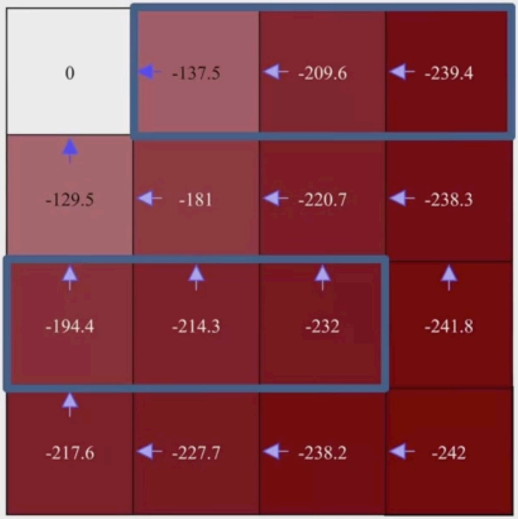


Policy Iteration

进行完 evaluation 后，得到每个状态的 value.

Evaluation

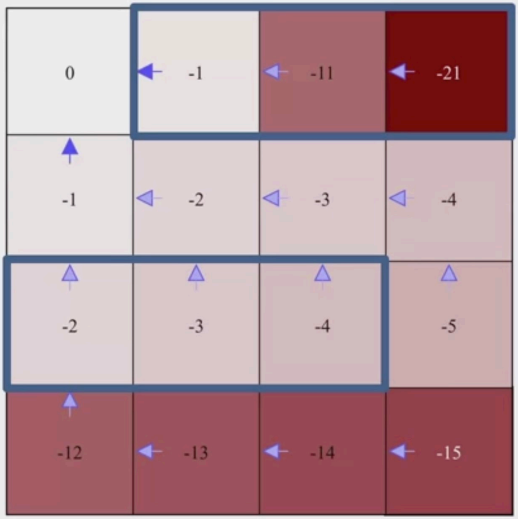
Improvement



Policy Iteration

Evaluation

Improvement

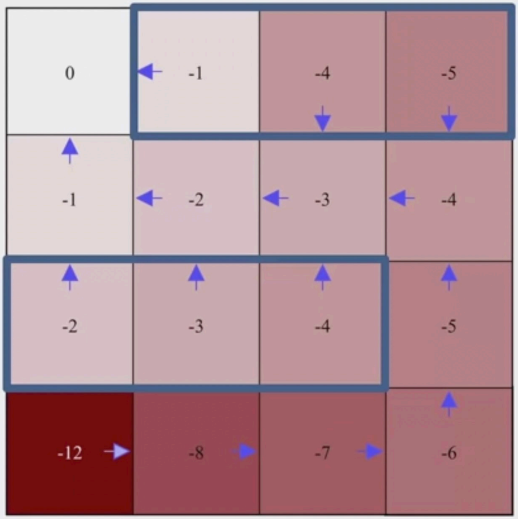


Policy Iteration

Evaluation

Improvement

6:21 / 8:22

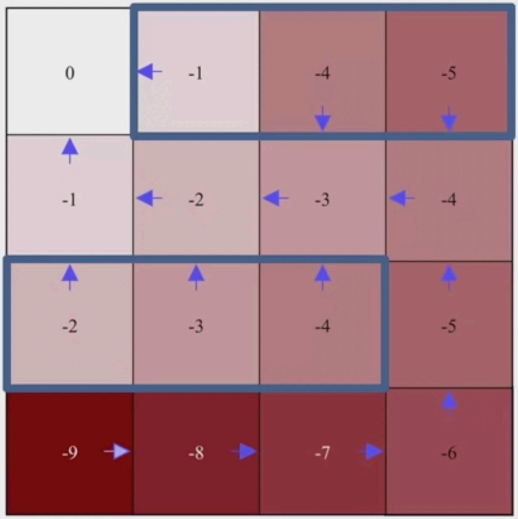


Policy Iteration

Evaluation

Improvement

→ 第3次 iteration 后, the policy is optimal and follows the low cost path avoid the blue state.



Policy Iteration

Evaluation

Improvement

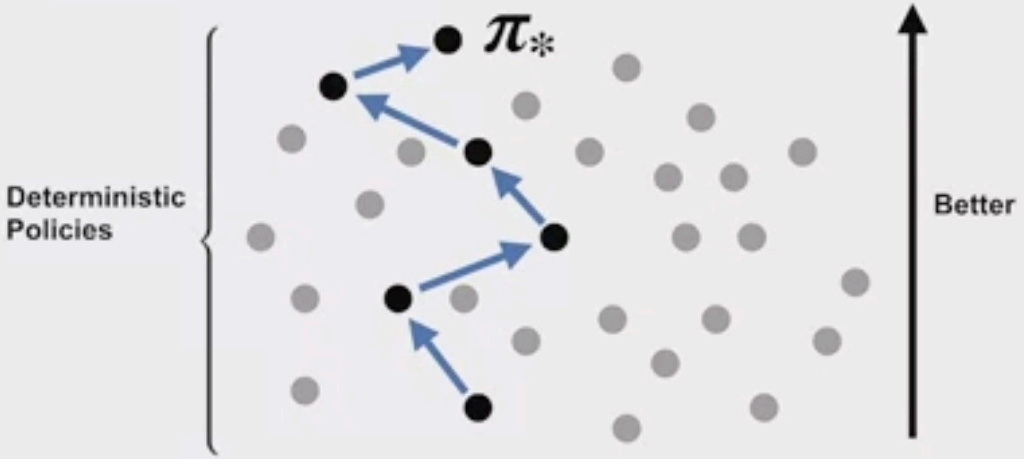
→ Evaluate again since the optimal value function.

If we greedified again, the policy remains unchanged.

∴ The policy iteration has been complete, and the optimal policy has been found.

The Power of Policy Iteration

=> we can follow a sequence of increasingly better policies until we reach an optimal policy.



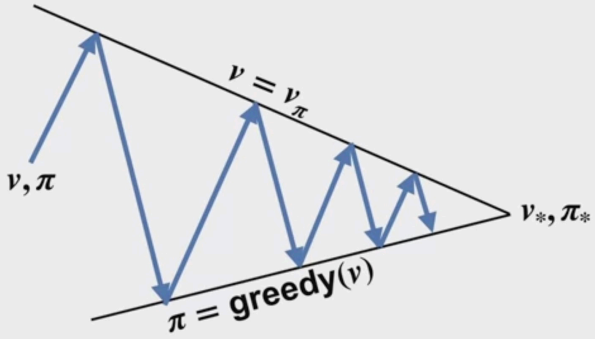
Summary

- Policy Iteration works by alternating policy evaluation and policy improvement
- Policy Iteration follows a sequence of better and better policies and value functions until it reaches the optimal policy and associated optimal value function

到目前为止，我们已经把政策迭代作为一个相当严格的程序来介绍。我们在评估当前政策和贪婪地改进政策之间交替进行。广义策略迭代的框架允许比这更多的自由，同时保持我们的最优性保证。在本视频中，我们将概述其中的一些替代方案。

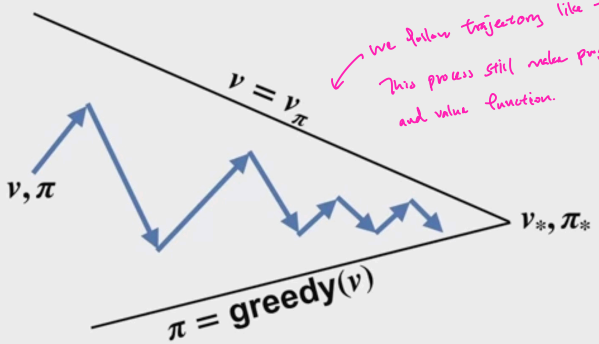
回顾政策和价值的舞蹈(dance of policy and value)。政策迭代算法一直运行到完成的每一步。直观地说，我们可以想象放宽这一点。想象一下，相反，我们遵循这样的一个轨迹。每个评估步骤都使我们的估计值稍微接近当前政策的价值，但不是全部。每个政策改进步骤使我们的政策更贪婪一点，但不是完全贪婪。直观地说，这个过程仍然应该朝着最优政策和价值函数的方向取得进展。事实上，理论告诉我们同样的事情。我们将使用广义政策迭代这个术语来指代我们可以交错进行政策评估和政策改进的所有方式。这给我们带来了我们的第一个广义政策迭代算法，称为价值迭代。在价值迭代中，我们仍然扫过所有的状态，并对当前的价值函数进行贪婪化。然而，我们并没有将策略评估运行到完成。我们只对所有的状态进行一次扫描。在此之后，我们再次进行贪婪化。我们可以把这写成一个直接适用于状态值函数的更新规则。这个更新并不参考任何特定的策略，因此被称为价值迭代。完整的算法看起来与迭代策略评估非常相似。我们不是根据一个固定的假说来更新价值，而是使用使当前价值估计最大化的行动来更新。价值迭代在极限情况下仍然收敛于V星。我们可以通过取argmax从最优价值函数中恢复出最优策略。在实践中，我们需要指定一个终止条件，因为我们不可能永远等待。我们将使用我们用于策略评估的相同条件。我们只需在值函数在整个扫频过程中的最大变化小于某个小值Theta时终止。值迭代在每次迭代中都会对整个状态空间进行扫描，就像策略迭代一样。像这样进行系统扫描的方法被称为同步方法。如果状态空间很大，这可能会有问题。每次清扫都会花费很长的时间。异步动态编程算法以任何顺序更新状态的值，它们不执行系统的扫描。它们可能会在另一个状态被更新之前多次更新一个给定的状态，甚至是一次。

Policy Iteration



Generalized Policy Iteration

→ all the ways we can interleave policy evaluation and policy improvement.

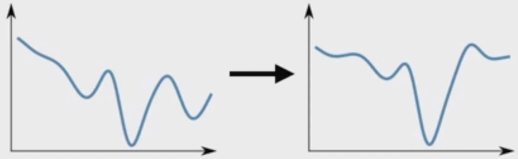


we follow trajectory like this
This process still makes progress towards the optimal policy and value function.

Imagine instead, we follow a trajectory like this. Each evaluation step brings our estimate a little closer to the value of the current policy but not all the way. Each policy improvement step makes our policy a little more greedy, but not totally greedy.

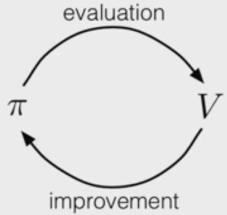
Value Iteration

Evaluation Step: to perform just one sweep over all states.



Improvement Step: then after that, we greedily again.

$$\pi \leftarrow \text{greedy}(v)$$



Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ => termination condition (terminate when the max change of the value function of the whole sweep <)

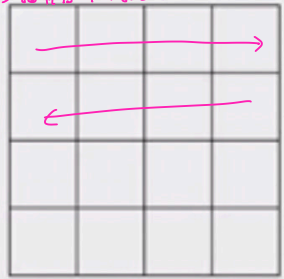
Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

After that, we greedify again. We can write this as an update rule which applies directly to the state value function. The update does not reference any specific policy, hence the name value iteration. The full algorithm looks very similar to iterative policy evaluation. Instead of updating the value according to a fixed falsey, we update using the action that maximizes the current value estimate. Value iteration still converges to V star in the limit. We can recover the optimal policy from the optimal value function by taking the argmax.

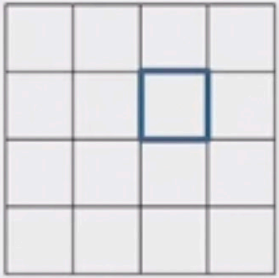
Avoiding full sweeps

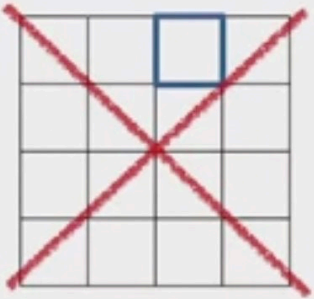
update the value of state in any order
do not perform systematic sweep
may update a given state many times before another is updated even once.

Synchronous DP



Asynchronous DP





为了保证收敛性，异步算法必须继续更新所有状态的值。例如，这里的算法永远更新相同的三个状态，而忽略其他所有的状态。这是不可接受的，因为如果其他状态根本没有被更新，那么它们就不可能是正确的。异步算法可以通过选择性的更新快速传播价值信息。有时这比系统性的扫描更有效。例如，一个异步方法可以更新那些最近改变了价值的状态附近的状态。这段视频就到此为止。你现在应该明白，价值迭代使我们能够在一个步骤中结合策略评估和改进，异步动态编程方法使我们能够自由地以任何顺序更新状态，而广义策略迭代的想法统一了许多算法。这包括值迭代、异步动态编程，以及你在本专业中将要涉及的几乎所有方法。

Summary

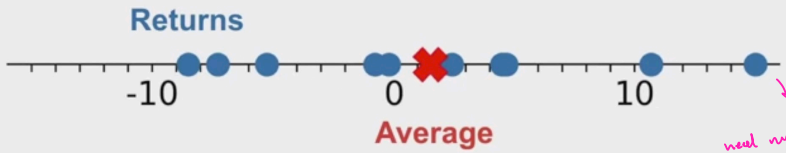
- Value Iteration allows us to combine policy evaluation and improvement into a single update
- Asynchronous dynamic programming methods give us the freedom to update states in any order
- Generalized Policy Iteration unifies classical DP methods, value iteration, and asynchronous DP

Iterative policy evaluation is the dynamic programming solution to the prediction or policy evaluation problem.

A Sampling Alternative for Policy Evaluation

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

Monte Carlo method



need many returns. Each return might be very different from the true state value. ∴ we may need to average many returns before true estimate converges. And we have to do this for every single state.

The value of each state can be treated as a totally independent estimation problem.

First, recall that the value is the expected return from a given state.

$V_\pi(s)$

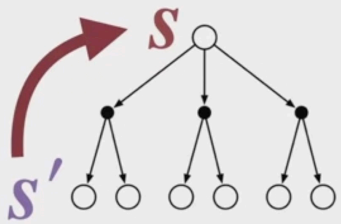
The procedure is simple, first, we gather a large number of returns under pi and take their average.

This will eventually converge to the state value,

this is called the Monte Carlo method.

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_k(s')\right]$$

Bootstrapping



each value independently.

The key insight of dynamic programming is that we do not have to treat the evaluation of each state as a separate problem. We can use the other value estimates we have already worked so hard to compute.

This process of using the value estimates of successor states to improve our current value estimate is known as bootstrapping. This can be much more efficient than a Monte Carlo method that estimates each value independently.

Brute-Force Search

$\pi(s_1)$	$\pi(s_2)$	$\cdots$	$\pi(s_{\lvert\mathcal{S}\rvert})$
a_3	a_1	$\cdots$	a_2
a_4	a_2	$\cdots$	a_3
		$\cdots$	
a_2	a_7	$\cdots$	a_3

Policy iteration computes optimal policies, brute-force search is a possible alternative.

This method simply evaluates every possible deterministic policy one at a time, we then pick the one with the highest value. There are a finite number of deterministic policies, and there always exists an optimal deterministic policy. So brute-force search will find the answer eventually, however, the number of deterministic policies can be huge.

Brute-Force Search

$\pi(s_1)$	$\pi(s_2)$	$\cdots$	$\pi(s_{\lvert\mathcal{S}\rvert})$

$$\lvert\mathcal{A}\rvert^{\lvert\mathcal{S}\rvert} \left\{ \quad \lvert\mathcal{A}\rvert * \lvert\mathcal{A}\rvert * \cdots * \lvert\mathcal{A}\rvert \right.$$

Policy Improvement Theorem $\pi_1 \rightarrow \pi_2 \rightarrow \ldots \rightarrow \pi_*$

A deterministic policy consists of one action choice per state. So the total number of deterministic policies is exponential in the number of states. Even on a fairly simple problem, this number could be massive, this process could take a very long time. The policy improvement theorem guarantees that policy iteration will find a sequence of better and better policies. This is a significant improvement over exhaustively trying each and every policy.

Efficiency of Dynamic Programming

Policy Iteration: Polynomial time in $|\mathcal{S}|$ and $|\mathcal{A}|$

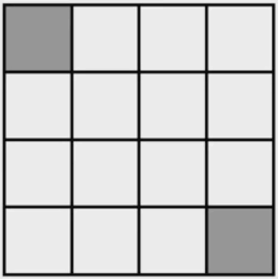
Exponentially faster

Brute-Force Search: $|\mathcal{A}|^{|\mathcal{S}|}$ policies

Well, policy iteration is guaranteed to find the optimal policy in time polynomial in the number of states and actions.

Thus, dynamic programming is exponentially faster than the brute-force search of the policy space.

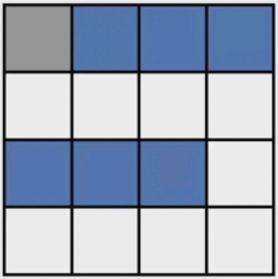
Dynamic Programming is much faster in practice



1 step

converged in just one
step of policy iteration.

Dynamic Programming is much faster in practice

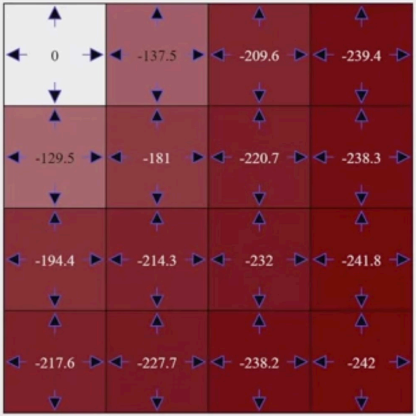


5 steps ⇒ 用DP只需迭代5次.

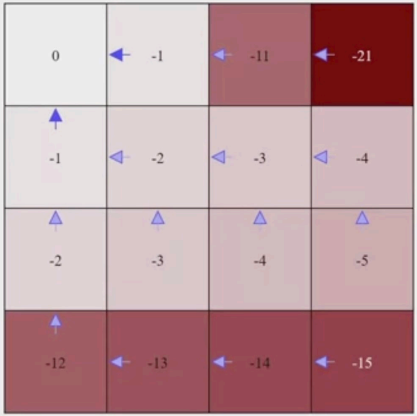
Brute-Force Search: 4^{16} policies ⇒ 用Brute-Force Search 需迭代 4^{16} 次.

In practice, this is not so bad, with each iteration,

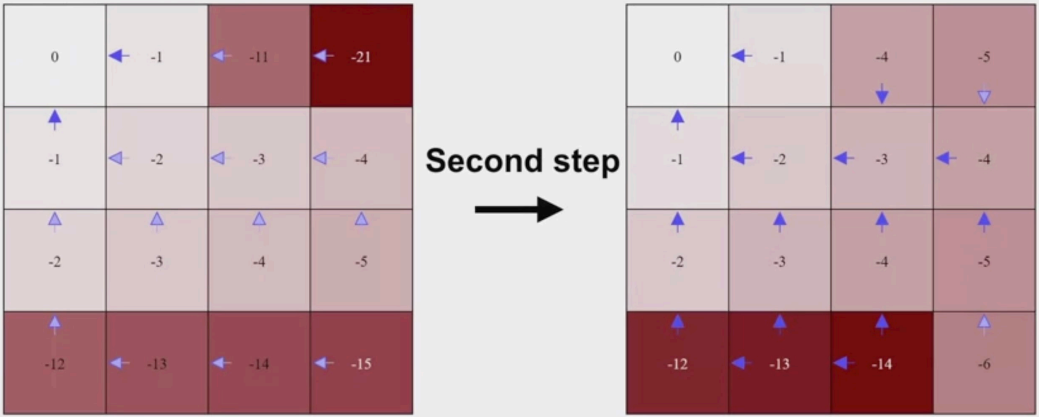
Dynamic Programming is much faster in practice



First step



Dynamic Programming is much faster in practice



The policy evaluation step changes
the value function less and

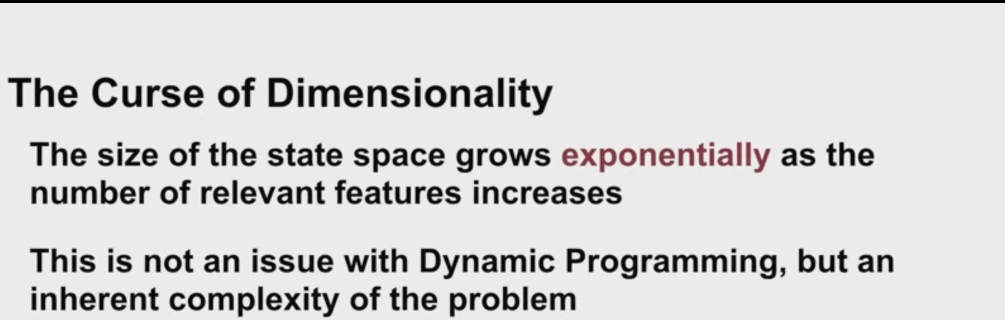
For example, the original four-by-four GridWorld converged in just one step of policy iteration.

When we made the problem harder by adding bad states, it still converged in just five iterations.

It might also seem restrictive that we have to run policy evaluation to completion for each step of policy iteration. In practice, this is not so bad, with each iteration, the policy tends to change less and less.

The Curse of Dimensionality

The size of the state space grows exponentially as the number of relevant features increases



the remainder of our time together,
that's it for this video.

Summary

- Bootstrapping can save us from performing a huge amount of unnecessary work

ADP for fleet optimization

$\hat{V}$: a vector. the marginal value of each of those drivers.

Step 1: Start with a pre-decision state S_t^n

Step 2: Solve the deterministic optimization using an approximate value function:

$$\tilde{v}_t^n = \min_x \left(C_t(S_t^n, x_t) + \bar{V}_t^{n-1}(S^{M,x}(S_t^n, x_t)) \right)$$

to obtain x^n.

Deterministic optimization

Step 3: Update the value function approximation

$$\bar{V}_{t-1}^n(S_{t-1}^{x,n}) = (1-\alpha_{n-1})\bar{V}_{t-1}^{n-1}(S_{t-1}^{x,n}) + \alpha_{n-1}\tilde{v}_t^n$$

Recursive statistics

=> update the marginal values $\bar{V}$

Step 4: Obtain Monte Carlo sample of W_t and compute the next pre-decision state:

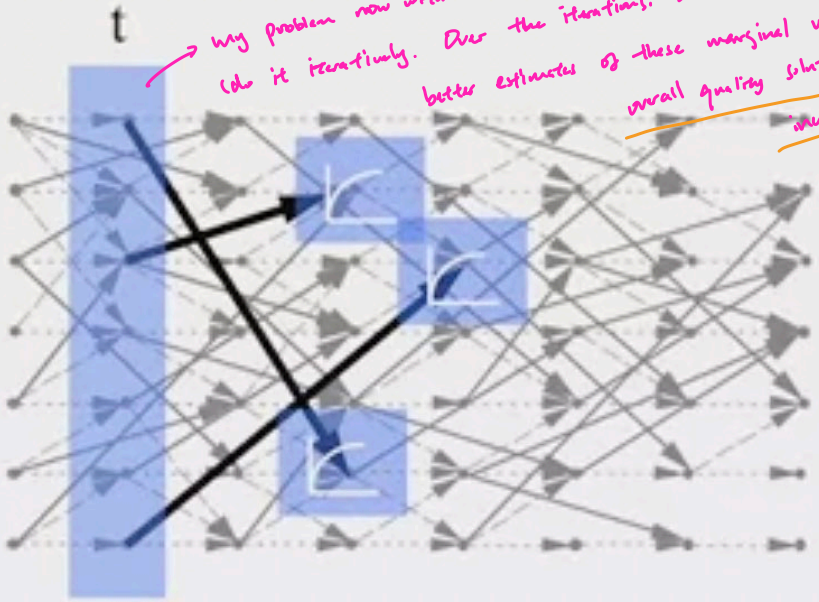
$$S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}(\omega^n))$$

Simulation

=> simulate forward in time.

Step 5: Return to step 1.

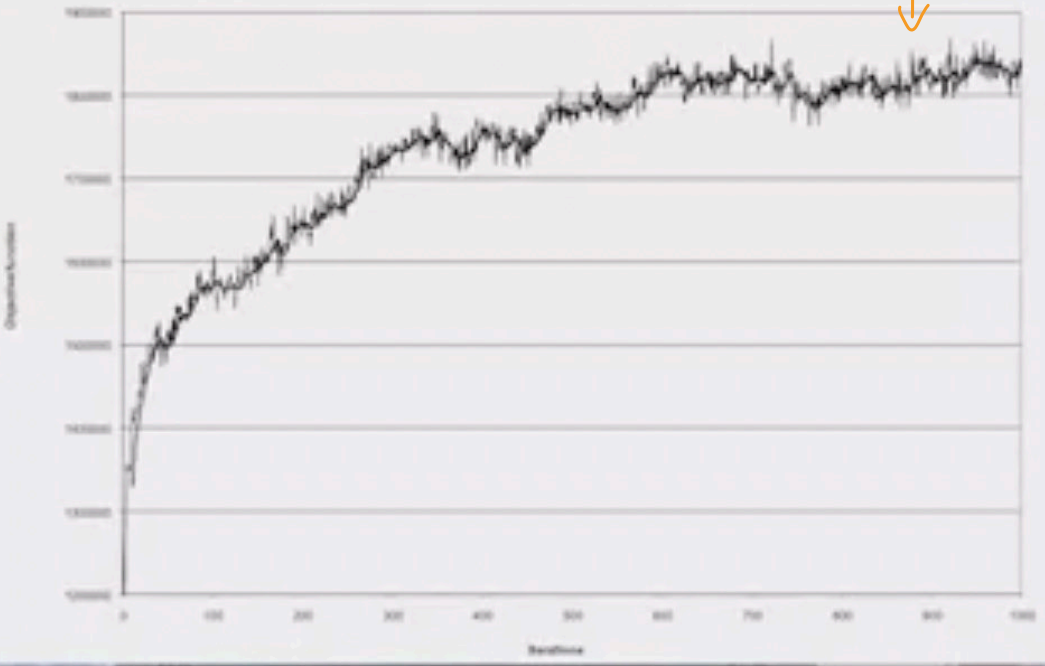
ADP for fleet optimization



My problem now with a value in the future.
(do it iteratively. Over the iterations, I'm going to get better and better estimates of these marginal values. and the overall quality solution tends to increase not uniformly.

ADP for fleet optimization

- … a typical performance graph.



本周，我们学习了所有关于动态编程的知识，以及如何用它来解决政策评估和控制的任务。这些算法构成了我们以后要学习的强化学习算法的基础。在这段视频中，我们将对我们所涉及的一切做一个快速回顾。

政策评估（Policy evaluation）是确定特定政策pi的错误价值函数（mistake value function）v_pi的任务。迭代策略评估（Iterative policy evaluation）将v_pi的贝尔曼方程变成一个更新规则。它产生一连串对v pi的更好的近似。控制（Control）是指改进策略的任务。政策改进定理告诉我们如何从一个给定的政策中构建一个更好的政策。新的策略pi' 是通过简单地对当前值进行greifying而产生的。Pi' 被保证严格优于pi，除非pi已经是最优的。

控制的动态编程算法（dynamic programming algorithm for control）是建立在策略改进定理之上的。这种算法被称为策略迭代（policy iteration）。政策迭代包括两个步骤：政策评估（policy evaluation）和政策改进（policy improvement）。在政策评估中，我们找到当前政策的价值函数。政策改进或贪婪化，即让政策在当前的价值函数方面变得贪婪。这些步骤反复进行，直到政策不发生变化。在这种情况下，政策保证是最优的。

Policy evaluation is the task of determining the state-value function v_π, for a particular policy π

produces

Iterative Policy Evaluation

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

$$\downarrow$$

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_k(s')\right]$$

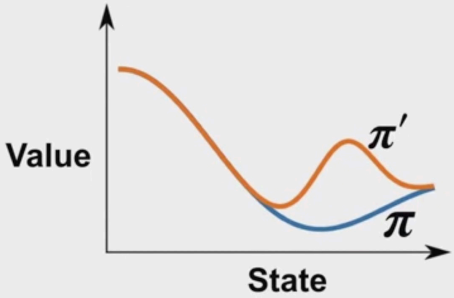
better and better approximations to v pi.

Control refers to the task of improving a policy

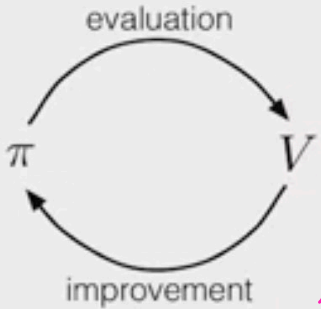
Policy improvement theorem

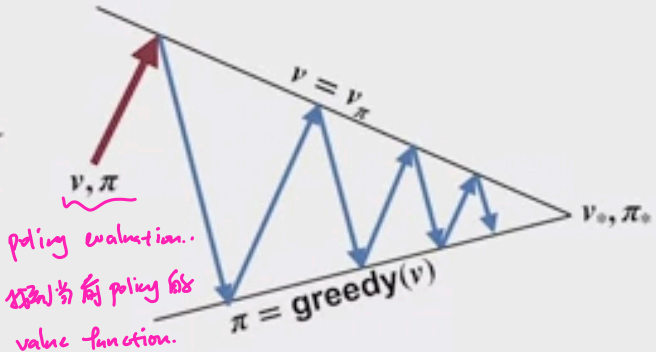
$$\pi'(s) \doteq \operatorname*{argmax}_a \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]$$

$\pi' > \pi$ unless π is optimal



Policy Iteration

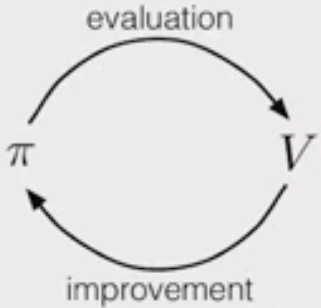


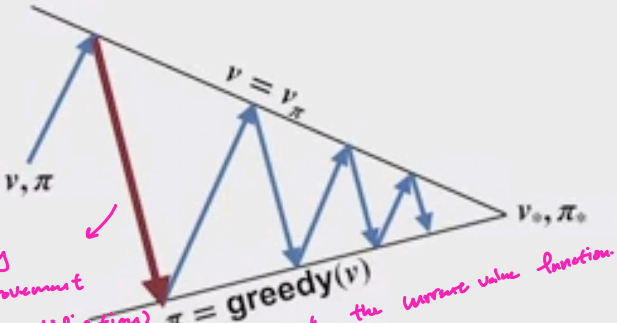


policy evaluation..

逐渐为每个 policy 的 value function.

Policy Iteration



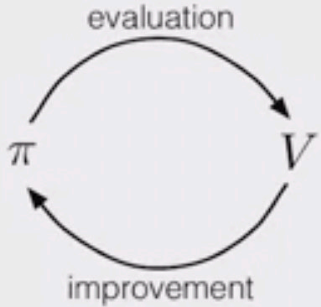


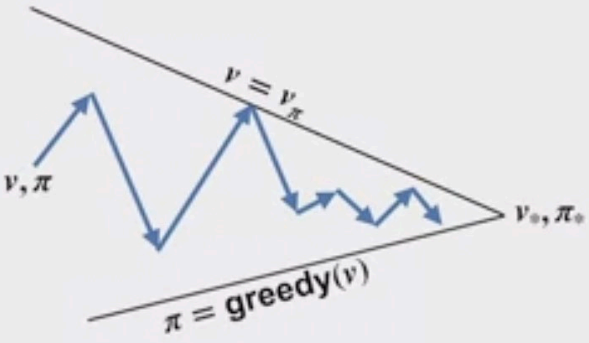
policy improvement (PI greedification)

makes the policy greedy with respect to the current value function.

我们还讨论了广义政策迭代（generalized policy iteration）的框架。在广义的政策迭代中，评估和改进步骤不需要运行到完成。广义政策迭代的一个例子是价值迭代（value iteration）。广义策略迭代还包括异步动态编程方法（asynchronous dynamic programming methods）。同步方法重复地扫过整个状态空间。异步方法更加灵活，它们可以以任何顺序更新状态。异步方法可以更有效地传播价值信息。当状态空间非常大时，这可能特别有用。异步方法可以被设计为专注于少数相关状态。在动态编程中，我们在某种意义上假设了可能的最佳情况。我们清楚地知道NDP是如何工作的，然而它仍然需要相当多的思考和聪明的算法来有效地计算出最佳政策。在本课程剩下的时间里，我们将在几乎所有的算法中继续收获这种聪明的好处。

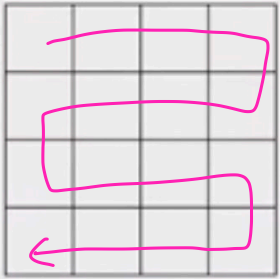
Generalized Policy Iteration



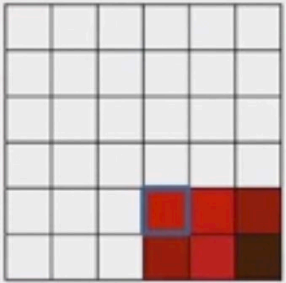


Generalized Policy Iteration includes asynchronous dynamic programming methods

Synchronous → 按顺序更新



Asynchronous → Q-L按任何顺序更新



Dynamic Programming

最新提交作业的评分 100%

1. The value of any state under an optimal policy is ___ the value of that state under a non-optimal policy. [Select all that apply] **1 / 1 分**

- [ ] Strictly greater than
- [x] Greater than or equal to

> **正确**
> Correct! This follows from the policy improvement theorem.

- [ ] Strictly less than
- [ ] Less than or equal to

2. If a policy π is greedy with respect to its own value function v_π, then it is an optimal policy. **1 / 1 分**

- (•) True
- ( ) False

> **正确**
> Correct! If a policy is greedy with respect to its own value function, it follows from the policy improvement theorem and the Bellman optimality equation that it must be an optimal policy.

3. Let v_π be the state-value function for the policy π. Let π' be greedy with respect to v_π. Then $v_{\pi'} \geq v_\pi$. 1 / 1 分

- ○ False
- ◉ True

正确
Correct! This is a consequence of the policy improvement theorem.

4. What is the relationship between value iteration and policy iteration? [Select all that apply] 1 / 1 分

- ☐ Policy iteration is a special case of value iteration.
- ☐ Value iteration is a special case of policy iteration.
- ☑ Value iteration and policy iteration are both special cases of generalized policy iteration.

正确
Correct!

5. The word synchronous means "at the same time". The word asynchronous means "not at the same time". A dynamic programming algorithm is: [Select all that apply]

1 / 1 分

- [x] Asynchronous, if it updates some states more than others.

正确
Correct! Only algorithms that update every state exactly once at each iteration are synchronous.

- [x] Synchronous, if it systematically sweeps the entire state space at each iteration.

正确
Correct! Only algorithms that update every state exactly once at each iteration are synchronous.

- [x] Asynchronous, if it does not update all states at each iteration.

正确
Correct! Only algorithms that update every state exactly once at each iteration are synchronous.

6. Policy iteration and value iteration, as described in chapter four, are synchronous.

1 / 1 分

- (•) True
- ( ) False

正确
Correct! As described in lecture, policy iteration and value iteration update all states systematic sweeps.

7. Which of the following is true? 1 / 1 分

- ○ Synchronous methods generally scale to large state spaces better than asynchronous methods.
- ◉ Asynchronous methods generally scale to large state spaces better than synchronous methods.

正确
Correct! Asynchronous methods can focus updates on more relevant states, and update less relevant states less often. If the state space is very large, asynchronous methods may still be able to achieve good performance whereas even just one synchronous sweep of the state space may be intractable.

8. Why are dynamic programming algorithms considered planning methods? [Select all that apply] 1 / 1 分

- ☐ They learn from trial and error interaction.
- ☐ They compute optimal value functions.
- ☑ They use a model to improve the policy.

正确
Correct! This is the definition of a planning method.

1 / 1 分

9. Consider the undiscounted, episodic MDP below. There are four actions possible in each state, A = {up, down, right, left}, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. The right half of the figure shows the value of each state under the equiprobable random policy. If π is the equiprobable random policy, what is $q(7, \text{down})$?

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

R = -1 on all transitions

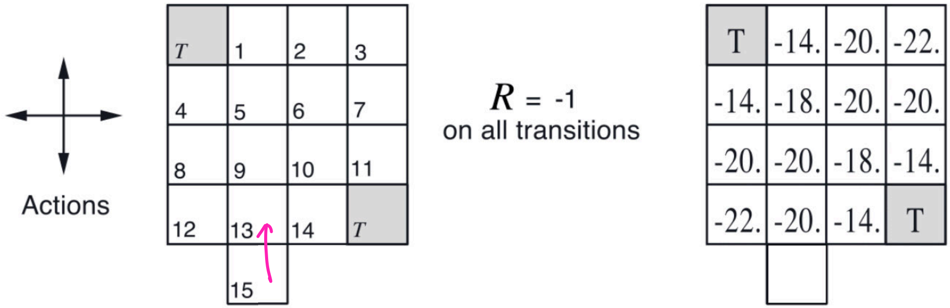
T	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	T

- $q(7, \text{down}) = -14$
- $q(7, \text{down}) = -20$
- $q(7, \text{down}) = -21$
- **$q(7, \text{down}) = -15$** ✓

正确

Correct! Moving down incurs a reward of -1 before reaching state 11, from which the expected future return is -14.

10. Consider the undiscounted, episodic MDP below. There are four actions possible in each state, A = {up, down, right, left}, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. The right half of the figure shows the value of each state under the equiprobable random policy. If π is the equiprobable random policy, what is $v(15)$? Hint: Recall the Bellman equation $v(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v(s')]$.



从 state 15 有 $\frac{1}{4}$ 的机率往上走, R=-1, 加上 state 13 的 value 为 -20.

$x = \frac{1}{4}(-1-20) + \frac{3}{4}(-1+x)$

有 $\frac{3}{4}$ 的机率会朝别的方向 (左, 右, 下) 走, R=-1, next state value 不变 (因为 state 15 的左, 右, 下 都没有东西). ∴将 next state value 设为 x.

$x = \frac{1}{4}(-21) - \frac{3}{4} + \frac{3}{4}x$

$x = -\frac{24}{4} + \frac{3}{4}x$

$x = -6 + \frac{3}{4}x$

$x - \frac{3}{4}x = -6$

$\frac{1}{4}x = -6$

$x = -24.$

- ○ $v(15) = -22$
- ◉ $v(15) = -24$
- ○ $v(15) = -23$
- ○ $v(15) = -21$
- ○ $v(15) = -25$

正确

Correct! We can get this by solving for the unknown variable $v(15)$. Let's call this unknown x. We solve for x in the equation $x = 1/4(-21) + 3/4(-1 + x)$. The first term corresponds to transitioning to state 13. The second term corresponds to taking one of the other three actions, incurring a reward of -1 and staying in state x.