动态规划（Dynamic Programming）

吉建民

USTC
jianmin@ustc.edu.cn

2023 年 9 月 24 日

Used Materials

Disclaimer: 本课件大量采用了 Rich Sutton's RL class, David Silver's Deep RL tutorial 和其他网络课程课件，也采用了 GitHub 中开源代码，以及部分网络博客内容

Table of Contents

What is Dynamic Programming?

Dynamic sequential or temporal component to the problem

Programming optimizing a "program", i.e. a policy

- c.f. linear programming

- A method for solving complex problems
- By breaking them down into subproblems
  - Solve the subproblems
  - Combine solutions to subproblems

Requirements for Dynamic Programming

Dynamic Programming is a very general solution method for problems which have two properties:

- Optimal substructure
  - *Principle of optimality* applies
  - Optimal solution can be decomposed into subproblems
- Overlapping subproblems
  - Subproblems recur many times
  - Solutions can be cached and reused
- Markov decision processes satisfy both properties
  - Bellman equation gives recursive decomposition
  - Value function stores and reuses solutions

动态规划

- 动态规划（Dynamic Programming）：通过把原问题分解为子问题的方式求解的一类方法。
  - 通过记住求解过的子问题来节省时间
- 动态规划适用的问题需要满足两个性质：
  - 最优子结构（Optimal Substructure）：整个问题的最优解可以通过求解子问题得到（通过子问题的最优解构造出全局最优解）
    - Bellman's Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.
  - 重叠子问题（Overlapping Subproblems）：
    - 子问题多次重复出现
    - 子问题的求解结果可以储存下来并再次使用
- MDPs 满足上述性质：
  - 贝尔曼方程给出递归分解方法
  - 值函数可以作为子问题的求解结果

贝尔曼方程和贝尔曼最优方程

- $V_\pi(s),\ Q_\pi(s,a),\ V^*(s),\ Q^*(s,a)$:

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

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

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

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

- 彼此之间关系:

$$V_\pi(s) = \sum_{a\in A} \pi(a \mid s) Q_\pi(s,a),$$

$$V^*(s) = \max_a Q^*(s,a),$$

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'\in S} P(s' \mid s,a) v_\pi(s'),$$

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'\in S} P(s' \mid s,a) V^*(s')$$

动态规划与强化学习

- 动态规划是强化学习算法的基础，但一般不能直接应用
  - 动态规划需要完整的 MDP 模型（Perfect Model），同时有较高的计算量
  - 强化学习可以看作在没有完整模型的条件下，用少量的计算尽量达到动态规划的效果
- 动态规划求解 MDPs
  - 预测（Prediction）：计算特定策略 π 的值函数 V_π
    - 输入：MDP (S, A, P, R) 和策略 π
    - 输出：值函数 V_π
  - 控制（Control）：
    - 输入：MDP (S, A, P, R)
    - 输出：最优值函数 V^* 或最优策略 π^*

Table of Contents

迭代策略评估（Iterative Policy Evaluation）

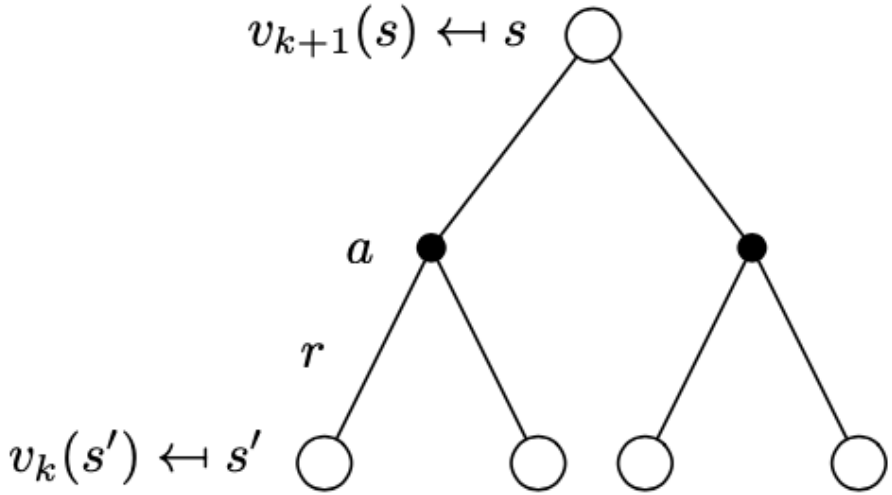
- 策略评估：给定策略 π，计算值函数 V_π
  - $V_\pi(s) = E_\pi\left(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s\right)$
  - $V_\pi(s) = \sum_a \pi(a \mid s)\left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_\pi(s')\right)$
- 迭代策略评估：给定策略 π，采用迭代的方式基于贝尔曼方程计算出其值函数 V_π
  - $V_1 \to V_2 \to \cdots \to V_\pi$
- 同步迭代算法：
  - 在第 $k+1$ 次迭代中，对所有状态 $s \in S$，基于 $V_k(s')$ 更新 $V_{k+1}(s)$，直到收敛
  - $V_\pi^{k+1}(s) = \sum_a \pi(a \mid s)\left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_\pi^k(s')\right)$

迭代策略评估算法

Iterative policy evaluation

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

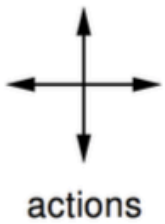
Iterative Policy Evaluation

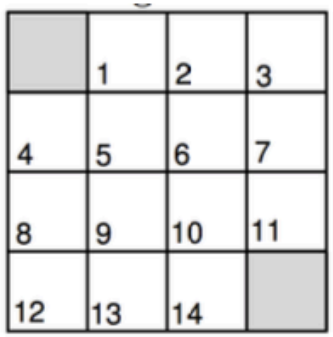


$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

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

Evaluating a Random Policy in the Small Gridworld



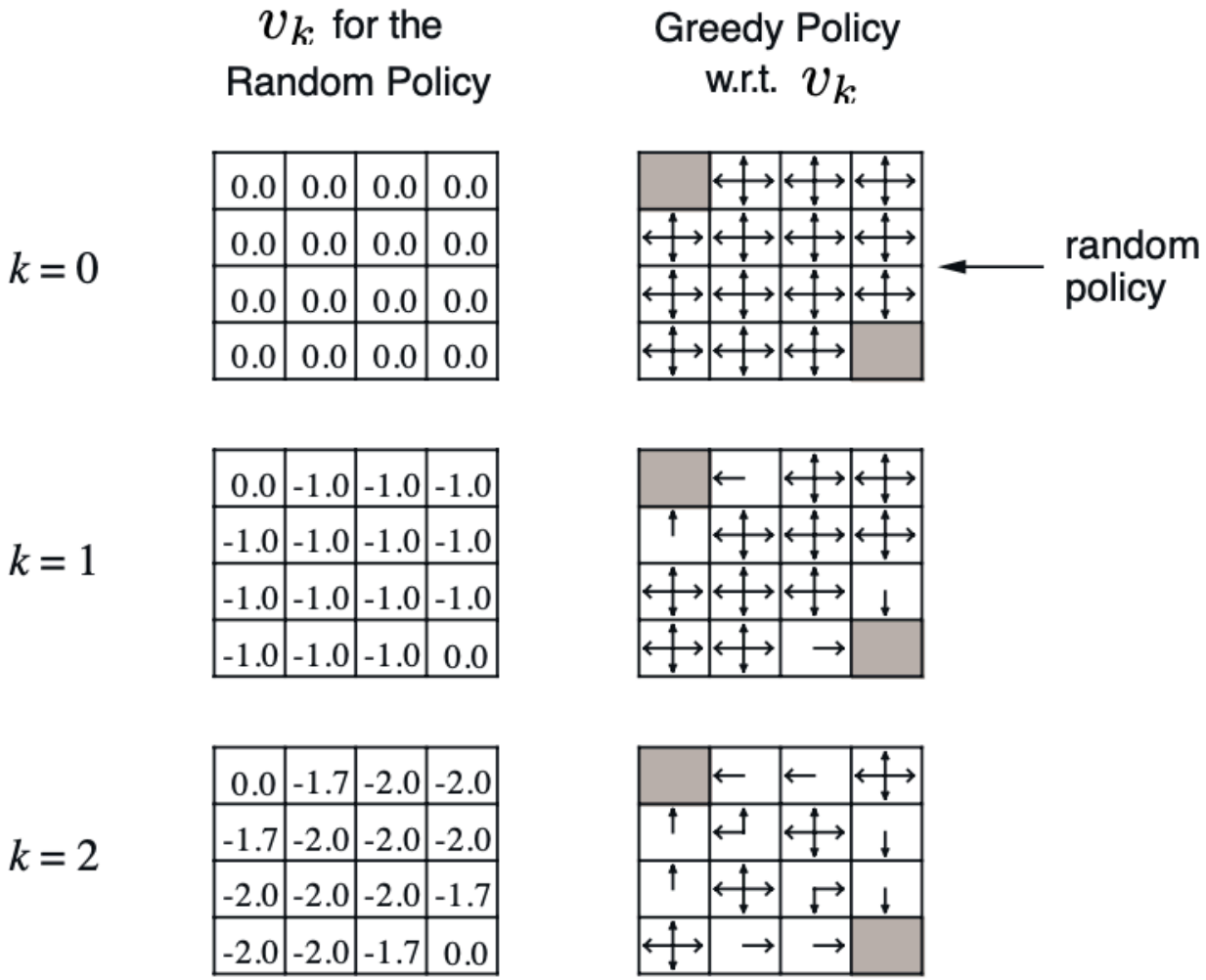


$r = -1$
on all transitions

- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states $1, \ldots, 14$
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Reward is -1 until the terminal state is reached
- Agent follows uniform random policy

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

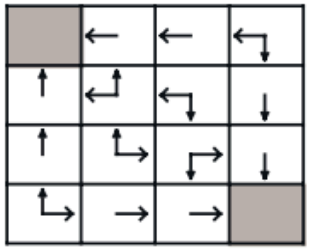
Iterative Policy Evaluation in Small Gridworld (1)



Iterative Policy Evaluation in Small Gridworld (2)

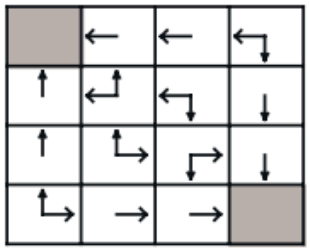
$k = 3$

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



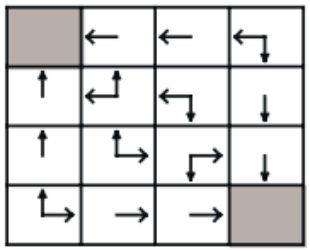
$k = 10$

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



$k = \infty$

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



optimal policy

策略评估直接求解

- 给定特定策略 π，MDP 变成 Markov Reward Process (MRP)

$$
\begin{aligned}
V_\pi(s) &= \sum_{a\in A}\pi(a\mid s)\left(R(s,a)+\gamma\sum_{s'}P(s'\mid s,a)V_\pi(s'\in S)\right)\\
&= \sum_{a\in A}\pi(a\mid s)R(s,a)+\gamma\sum_{a\in A}\pi(a\mid s)\sum_{s'\in S}P(s'\mid s,a)V_\pi(s')\\
&= R_s^\pi+\gamma\sum_{s'\in S}P_{s's}^\pi V_\pi(s')
\end{aligned}
$$

其中 $R_s^\pi=\sum_a\pi(a\mid s)R(s,a)$，$P_{s's}^\pi=\sum_a\pi(a\mid s)P(s'\mid s,a)$.

- 将所有状态 $s\in S$ 表达为向量，上式的矩阵形式为：

$$V_\pi = R^\pi+\gamma P^\pi V_\pi$$

直接求解结果为：

$$V_\pi=(I-\gamma P^\pi)^{-1}R^\pi$$

计算复杂性为 $O(N^3)$

Table of Contents

How to Improve a Policy

- Given a policy π
  - Evaluate the policy π

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

  - Improve the policy by acting greedily with respect to v_π

$$\pi' = \text{greedy}(v_\pi)$$

- In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
- In general, need more iterations of improvement / evaluation
- But this process of policy iteration always converges to $\pi*$

策略迭代

- 给定策略 π
  - 评估（Evaluate）策略 π，计算 V_π
    - 迭代策略评估

$$V_\pi^{k+1}(s) = \sum_a \pi(a \mid s) \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_\pi^k(s') \right)$$

    - 直接策略评估

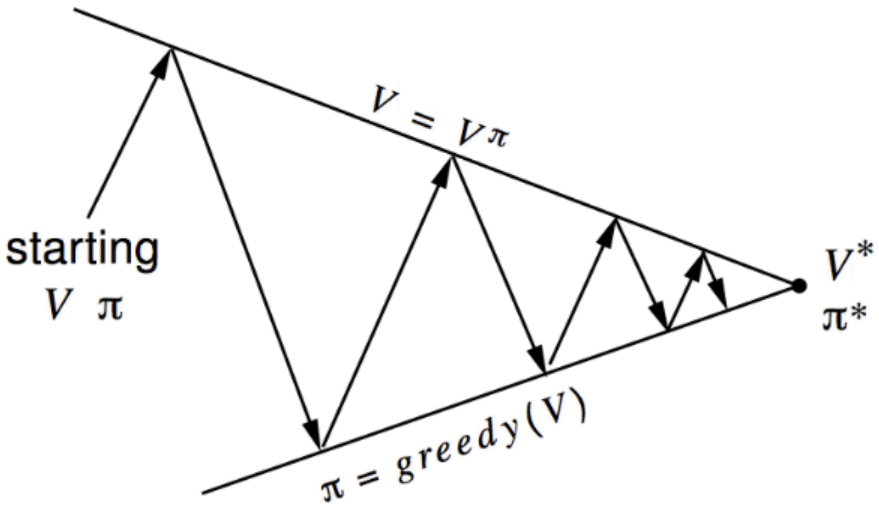
$$V_\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

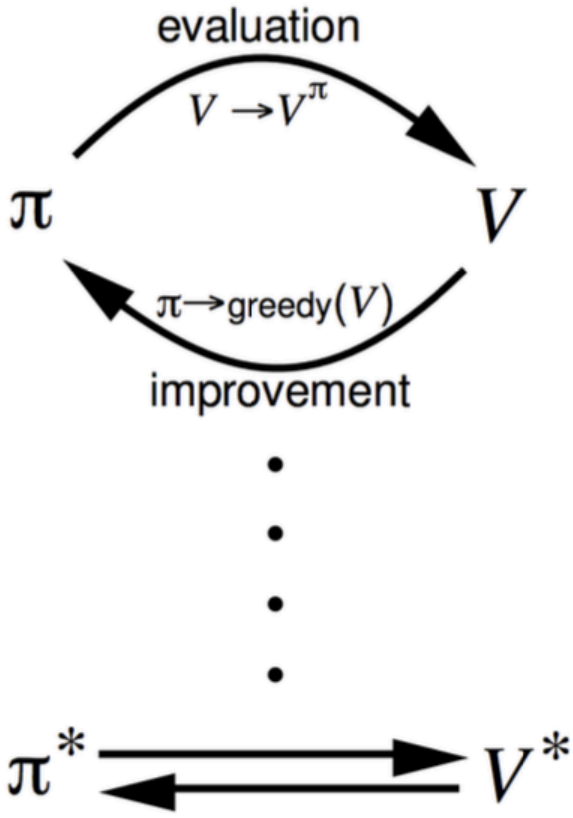
  - 改进（Improve）策略，基于 V_π 采用贪婪策略
    - $Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_\pi(s')$
    - 基于 $Q_\pi(s,a)$ 采用贪婪策略得到 π'

$$\begin{aligned} \pi'(s) &= argmax_a Q_\pi(s,a) \\ &= argmax_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_\pi(s') \right) \end{aligned}$$

  - 当策略不再改进，则已经是最优的。

Policy Iteration





Policy evaluation Estimate v_π

Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

Greedy policy improvement

策略迭代算法

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

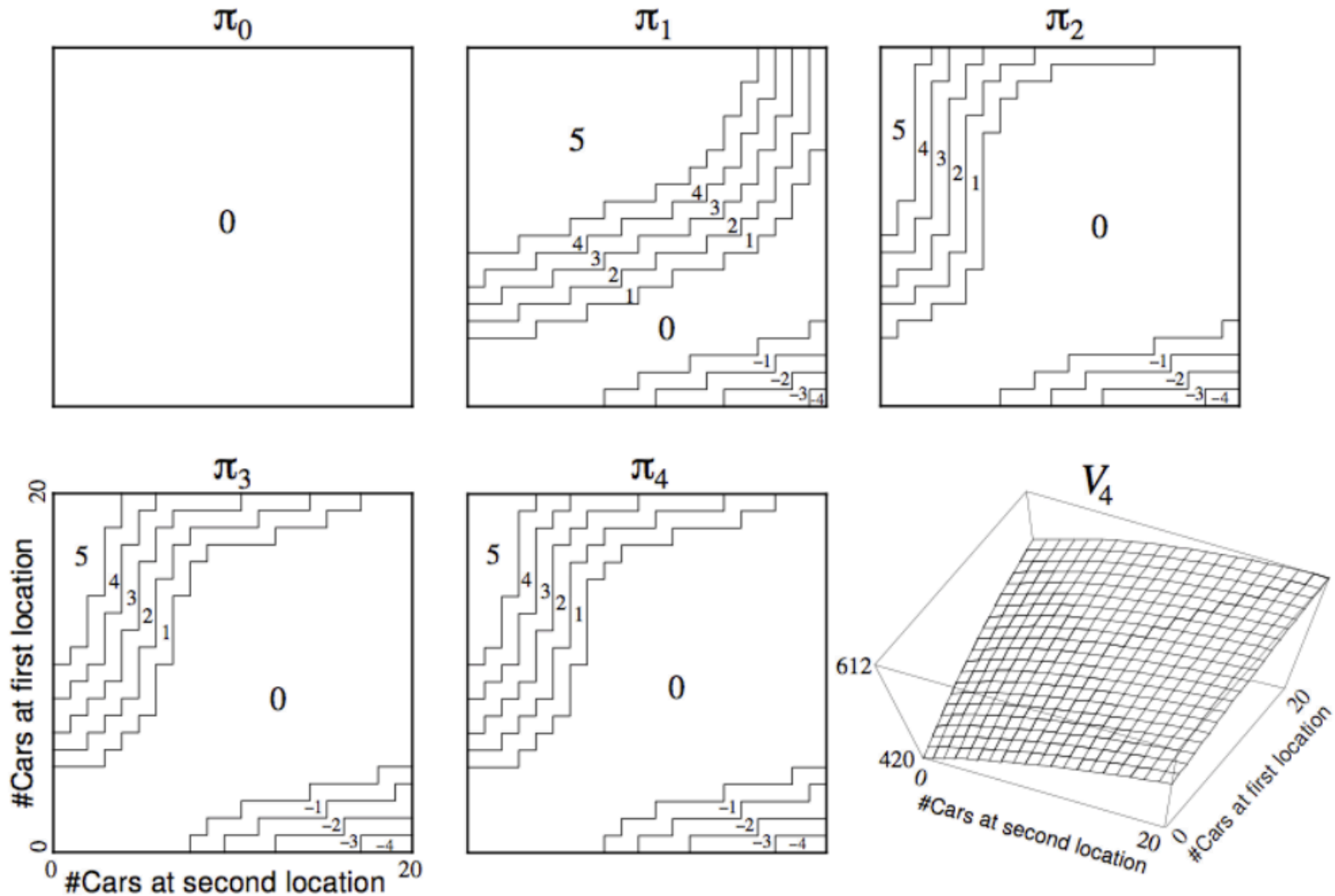
2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $\mathrm{v} \leftarrow V(s)$
   $V(s) \leftarrow \Sigma_{a \in \mathcal{A}} \pi(a|s) \left(r(s,a) + \gamma \Sigma_{s' \in \mathcal{S}} T(s'|s,a) V(s')\right)$
   $\Delta \leftarrow \max(\Delta, |\mathrm{v} - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $a \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max r(s,a) + \gamma \Sigma_{s' \in \mathcal{S}} T(s'|s,a) \mathrm{v}_\pi(s')$
   If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Jack's Car Rental

- States: Two locations, maximum of 20 cars at each
- Actions: Move up to 5 cars between locations overnight
- Reward: \$10 for each car rented (must be available)
- Transitions: Cars returned and requested randomly
  - Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!}e^{-\lambda}$
  - 1st location: average requests $= 3$, average returns $= 3$
  - 2nd location: average requests $= 4$, average returns $= 2$

Policy Iteration in Jack's Car Rental



Policy Improvement

- Consider a deterministic policy, $a = \pi(s)$
- We can *improve* the policy by acting greedily

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

- This improves the value from any state s over one step,

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

- It therefore improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$

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

Policy Improvement

- If improvements stop,

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

- Then the Bellman optimality equation has been satisfied

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

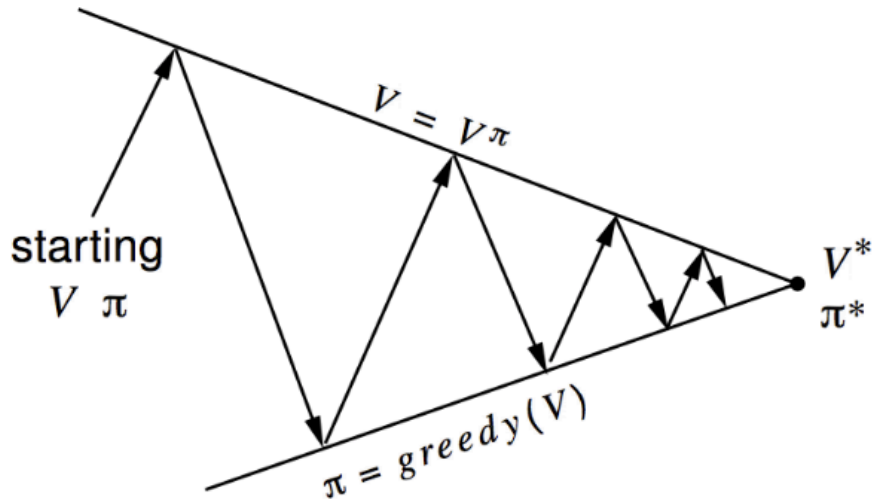
- Therefore $v_\pi(s) = v_*(s)$ for all $s \in \mathcal{S}$
- so π is an optimal policy

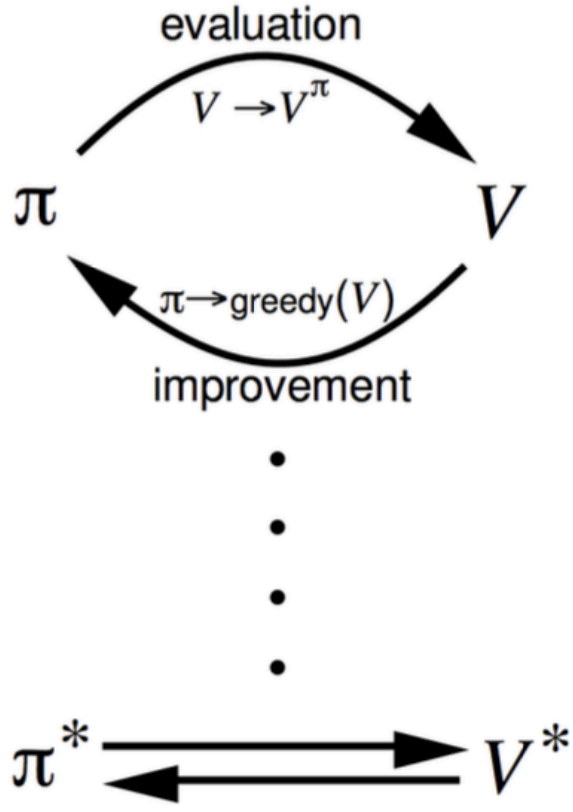
Modified Policy Iteration

- Does policy evaluation need to converge to v_π?
- Or should we introduce a stopping condition
  - e.g. ϵ-convergence of value function
- Or simply stop after k iterations of iterative policy evaluation?
- For example, in the small gridworld $k = 3$ was sufficient to achieve optimal policy
- Why not update policy every iteration? i.e. stop after $k = 1$
  - This is equivalent to *value iteration* (next section)

广义策略迭代（Generalized Policy Iteration）

- Generalized Policy Iteration (GPI): any interleaving of policy evaluation and policy improvement, independent of their granularity.





Policy evaluation Estimate v_π

Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$

Any policy improvement algorithm

Table of Contents

Principle of Optimality

Any optimal policy can be subdivided into two components:

- An optimal first action A_*
- Followed by an optimal policy from successor state S'

Theorem (Principle of Optimality)

A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if

- *For any state s' reachable from s*
- *π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$*

Deterministic Value Iteration

- If we know the solution to subproblems $v_*(s')$
- Then solution $v_*(s)$ can be found by one-step lookahead

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

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards
- Still works with loopy, stochastic MDPs

值迭代（Value Iteration）

- 当已知所有子问题的解，$V^*(s')$，则最终结果 $V^*(s)$ 可以通过下面公式计算

$$V^*(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s') \right)$$

- 值迭代算法就是将上述公式转换为迭代形式：

$$V^*_{k+1}(s) \leftarrow \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*_k(s') \right)$$

- 基于 $V^*(s)$ 可以计算出最优策略 π^*

$$\pi^*(s) = argmax_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s') \right)$$

值迭代算法

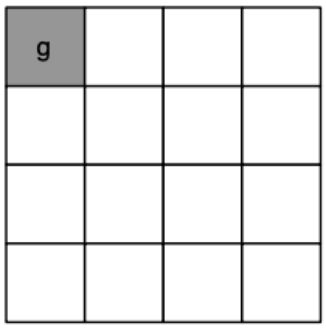
Value iteration

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

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

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

Example: Shortest Path



Problem

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

V_1

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

V_2

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

V_3

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

V_4

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

V_5

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

V_6

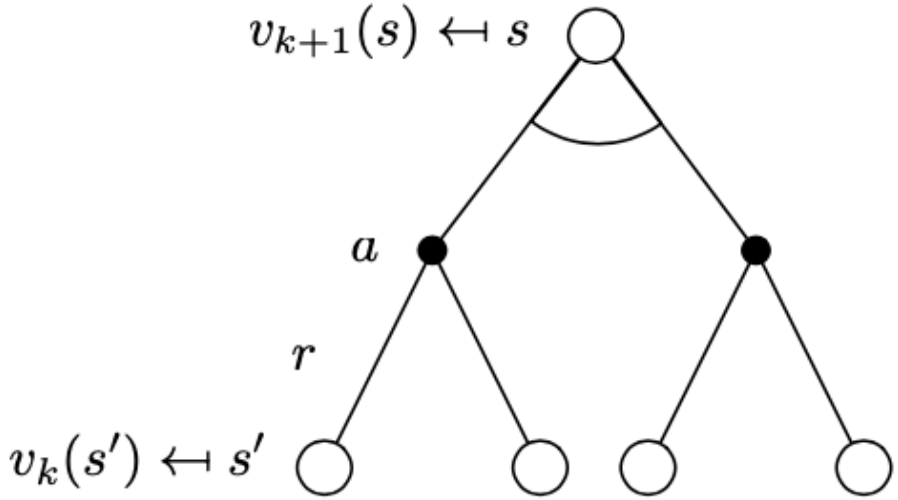
0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

Value Iteration

- Problem: find optimal policy π
- Solution: iterative application of Bellman optimality backup
- $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_*$
- Using synchronous backups
  - At each iteration $k+1$
  - For all states $s \in \mathcal{S}$
  - Update $v_{k+1}(s)$ from $v_k(s')$
- Convergence to v_* will be proven later
- Unlike policy iteration, there is no explicit policy
- Intermediate value functions may not correspond to any policy

Value Iteration (2)



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

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

策略迭代与值迭代的区别

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

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $temp \leftarrow v(s)$
         $v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s)) \Big[r(s, \pi(s), s') + \gamma v(s') \Big]$
         $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$ (a small positive number)

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

Figure 4.3: Policy iteration (using iterative policy evaluation) for v_*. This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it. :-)

finding optimal value function

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

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

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

Figure 4.5: Value iteration.

one policy update (extract policy from the optimal value function

同步（Synchronous）动态规划算法

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

- 迭代策略评估每步迭代的时间复杂性为 $O(mn^2)$，其中 m actions, n states
- 策略迭代每步迭代的时间复杂性为 $O(mn^2 + n^3)$，其中 policy improvement 用 $O(mn^2)$，policy evaluation 用 $O(n^3)$（求解线性方程的时间复杂度）
- 值迭代每步迭代的时间复杂性为 $O(mn^2)$
- 上述算法采用 state-value function $V_\pi(s)$ 或 $V^*(s)$，也可以采用 action-value function $Q_\pi(s,a)$ 或 $Q^*_\pi(s,a)$，值迭代每步的时间复杂性提高为 $O(m^2n^2)$

Table of Contents

异步（Asynchronous）动态规划

- 同步动态规划：all states are backed up in parallel
  - 当状态空间很大时，计算代价较大
  - 同步动态规划，目前能处理百万级别状态空间的问题
- 异步动态规划：backs up states individually, in any order
  - 可以很大的降低计算量
  - 当所有状态被持续选择，则保证收敛
- 异步动态规划算法：
  - 原位动态规划（In-place dynamic programming）
  - 优先扫描（Prioritized sweeping）
  - 实时动态规划（Real-time dynamic programming）

原位动态规划（In-place dynamic programming）

- 同步值迭代存储两份值函数

$$V_{new}(s) \leftarrow max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_{old}(s') \right)$$

$$V_{old}(s) \leftarrow V_{new}$$

- 原位动态规划只存储一份，随时更新

$$V(s) \leftarrow max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right)$$

- 原位动态规划即时计算即时更新。这样可以减少保存的状态价值的数量，节约内存。代价是收敛速度可能稍慢。

优先扫描（Prioritized sweeping）

- 优先级动态规划 (prioritised sweeping)：对每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，需要维护一个优先级队列。
- 贝尔曼误差（Bellman Error）

$$\left| \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right) - V(s) \right|$$

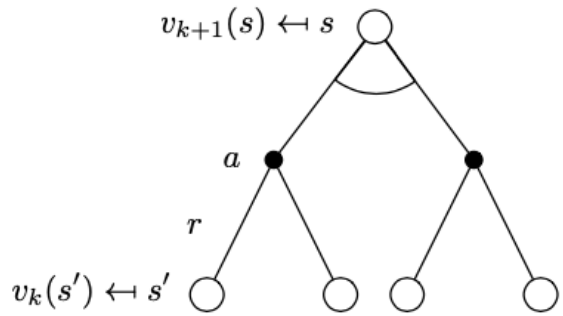
实时动态规划（Real-time dynamic programming）

- 实时动态规划 (real-time dynamic programming)：利用 Agent 与环境交互产生的实际经历来更新状态价值，选择 Agent 实际经历过的状态进行价值更新。
  - Agent 经常访问的状态将以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。收敛速度可能稍慢。
- 基本思想：只更新与 Agent 有关的状态，Use agent's experience to guide the selection of states
- 每步实际状态为 S_t，行动为 A_t，效用为 r_{t+1}，每步状态更新为：

$$V(S_t) \leftarrow max_a \left(R(S_t, a) + \gamma \sum_{s'} P(s' \mid S_t, a) V(s') \right)$$

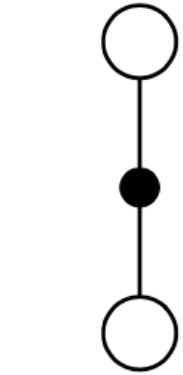
Full-Width Backups

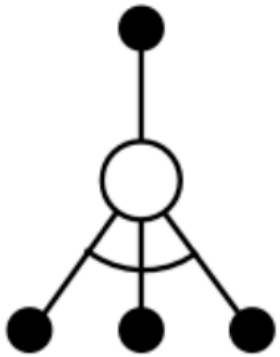
- DP uses *full-width* backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's *curse of dimensionality*
  - Number of states $n = |\mathcal{S}|$ grows exponentially with number of state variables
- Even one backup can be too expensive



Sample Backups

- In subsequent lectures we will consider *sample backups*
- Using sample rewards and sample transitions $\langle S, A, R, S' \rangle$
- Instead of reward function $\mathcal{R}$ and transition dynamics $\mathcal{P}$
- Advantages:
  - Model-free: no advance knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backup is constant, independent of $n = |\mathcal{S}|$





Approximate Dynamic Programming

- Approximate the value function
- Using a *function approximator* $\hat{v}(s, \mathbf{w})$
- Apply dynamic programming to $\hat{v}(\cdot, \mathbf{w})$
- e.g. Fitted Value Iteration repeats at each iteration k,
  - Sample states $\tilde{\mathcal{S}} \subseteq \mathcal{S}$
  - For each state $s \in \tilde{\mathcal{S}}$, estimate target value using Bellman optimality equation,

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

  - Train next value function $\hat{v}(\cdot, \mathbf{w_{k+1}})$ using targets $\{\langle s, \tilde{v}_k(s) \rangle\}$

Table of Contents

算法收敛性

- 迭代策略评估算法的结果是否收敛到 V_π？
- 策略迭代算法的结果是否收敛到 π^*？
- 值迭代算法的结果是否收敛到 V^*？
- 算法是否一定收敛，结果是否唯一，收敛速度多快？
- 收缩映射定理（Contraction Mapping Theorem）可以回答这些问题

收缩映射定理（Contraction Mapping Theorem）

- 范数（norm）：$||\vec{x}||$ 表示向量在向量空间中的长度
  - $||\vec{x}||_1 := \sum_{i=1}^{n} |x_i|$
  - $||\vec{x}||_2 := \sqrt{\sum_{i=1}^{n} x_i^2}$
  - $||\vec{x}||_p := \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$
  - $||\vec{x}||_\infty := \max_i |x_i|$
- An operator F on a normed vector space $\mathcal{X}$ is a γ-**contraction**, for $0 < \gamma < 1$, provided for all $x, y \in \mathcal{X}$

$$||F(x) - F(y)|| \leq \gamma ||x - y||$$

Theorem (Contraction Mapping Theorem)

For any metric space $\mathcal{V}$ that is complete (i.e. closed) under an operator $T(v)$, where T is a γ-contraction,

- *T converges to a unique fixed point*
- *At a linear convergence rate of γ*

值函数空间

- 考虑由所有可能值函数构成的向量空间 $\mathcal{V}$
- 其中每个向量 V 都是 $|S|$ 维
- 向量空间中每个点对应一个值函数 $V(\cdot)$
- 贝尔曼操作作为向量空间 $\mathcal{V}$ 中的操作
  - 贝尔曼期望操作（Bellman expectation backup operator）F^π

$$F^\pi(V) = R^\pi + \gamma P^\pi V$$

  - 贝尔曼最优操作（Bellman optimality backup operator）F^*

$$F^*(V) = \max_a (R(a) + \gamma P(a) V)$$

- 两个值函数 U 和 V 的距离由 ∞-norm 定义

$$||U - V||_\infty = \max_s |U(s) - V(s)|$$

- 这些贝尔曼操作都使得值函数更靠近（closer），从而收敛到唯一解

贝尔曼操作

- 贝尔曼期望操作是 γ-contraction

$$\begin{aligned} ||F^{\pi}(U) - F^{\pi}(V)||_{\infty} &= || \, (R^{\pi} + \gamma P^{\pi} U) - (R^{\pi} + \gamma P^{\pi} V) \, ||_{\infty} \\ &= ||\gamma P^{\pi}(U - V)||_{\infty} \\ &\leq ||\gamma P^{\pi}||U - V|_{\infty}||_{\infty} \\ &\leq \gamma||U - V|_{\infty} \end{aligned}$$

- 贝尔曼最优操作是 γ-contraction

$$||F^{*}(U) - F^{*}(V)||_{\infty} \leq \gamma||U - V|_{\infty}$$

收敛性结果

- 贝尔曼期望操作 F^{π} 和贝尔曼最优操作 F^{*} 各自有唯一的不动点
- V_{π} 是贝尔曼期望操作 F^{π} 的不动点
- 迭代策略评估算法收敛到 V_{π}，策略迭代算法收敛到 V^{*}
- V^{*} 是贝尔曼最优操作 F^{*} 的不动点
- 值迭代算法收敛到 V^{*}

小结

- 动态规划算法求解给定模型的 MDP 问题，是强化学习算法的基础
- 动态规划算法的核心是贝尔曼方程
- 策略评估算法，给定策略计算其相应的值函数
- 策略迭代算法，先评估策略，再改进策略，直到收敛
- 值迭代算法，采用贝尔曼最优操作不断迭代，直到收敛