Dynamic Programming

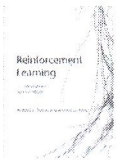
Machine Learning

Daniele Loiacono

POLITECNICO MILANO 1863

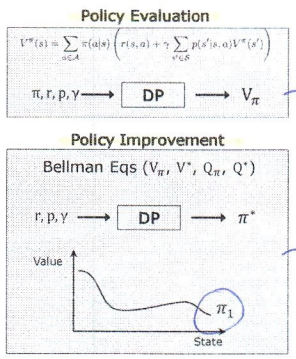
Outline and References

- Outline
  - Policy Evaluation
  - Policy Improvement
  - Policy Iteration
  - Generalized Policy Iteration
  - Efficiency of DP
- References
  - Reinforcement Learning: An Introduction [RL Chapter 4]
  - Fundamentals of Reinforcement Learning (Coursera)



Why Dynamic Programming?

- To solve an MDP we need to find an optimal policy
- Unfortunately we cannot use a brute-force approach:
  - $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies to evaluate
  - $|\mathcal{S}|$ linear equations to solve for each policy
- Dynamic Programming (DP) is a method that allow to solve a complex problem by breaking it down into simpler sub-problems in a **recursive** manner
- We will see how to use DP to solve an MDP thanks to the Bellman Equations

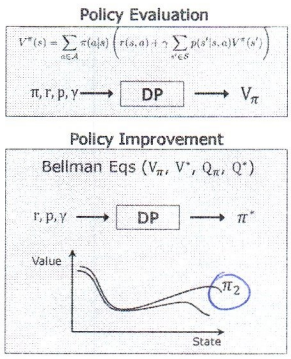


because of size of the state space we cannot provide using linear systems solver

thanks to the dynamic programming we'll improve iteratively the policy (from $\pi_1 \to \pi_2 \to \ldots \to \pi^*$)

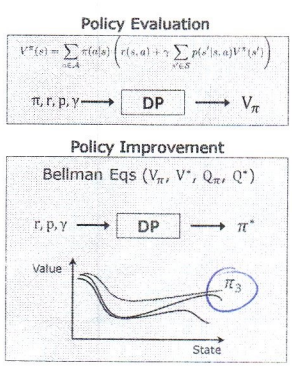
Why Dynamic Programming?

- To solve an MDP we need to find an optimal policy
- Unfortunately we cannot use a brute-force approach:
  - $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies to evaluate
  - $|\mathcal{S}|$ linear equations to solve for each policy
- Dynamic Programming (DP) is a method that allow to solve a complex problem by breaking it down into simpler sub-problems in a **recursive** manner
- We will see how to use DP to solve an MDP thanks to the Bellman Equations



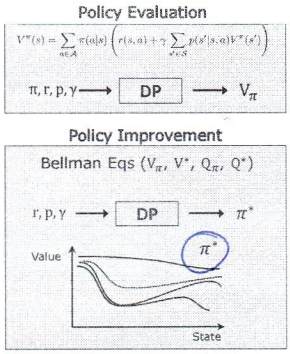
Why Dynamic Programming?

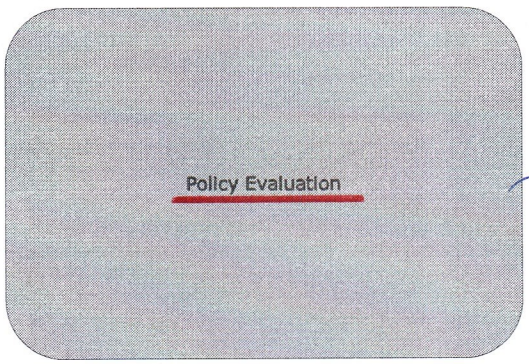
- To solve an MDP we need to find an optimal policy
- Unfortunately we cannot use a brute-force approach:
  - $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies to evaluate
  - $|\mathcal{S}|$ linear equations to solve for each policy
- Dynamic Programming (DP) is a method that allow to solve a complex problem by breaking it down into simpler sub-problems in a **recursive** manner
- We will see how to use DP to solve an MDP thanks to the Bellman Equations



Why Dynamic Programming?

- To solve an MDP we need to find an optimal policy
- Unfortunately we cannot use a brute-force approach:
  - $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies to evaluate
  - $|\mathcal{S}|$ linear equations to solve for each policy
- Dynamic Programming (DP) is a method that allow to solve a complex problem by breaking it down into simpler sub-problems in a **recursive** manner
- We will see how to use DP to solve an MDP thanks to the Bellman Equations





we have a given policy π, we need to evaluate it

Iterative Policy Evaluation

- We search the solution of the Bellman expectation equation:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \right)$$

- DP solves this problem through iterative application of Bellman equation:

$$V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \right) \quad *$$

  - At each iteration k, the value-function V_k is updated for all state $s \in \mathcal{S}$

$$V_0 \mapsto V_1 \rightarrow \cdots \rightarrow V_k \rightarrow V_{k+1} \rightarrow V_\pi$$ sweep

  - It can be proved that V_k converge to V_π as $k \rightarrow \infty$ for any V_0

initialization of the value function

Iterative Policy Evaluation (2)

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

Input π, the policy to be evaluated
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 $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

«in place» update

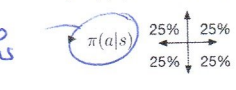
the alternative is:
old:
new:

we use the vector V(s) to update itself:

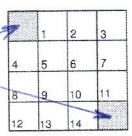
This brings the property that the final result depends on the order of the spaces in the vector V(·). Moreover, this cannot be parallelized. However, it converges fast.

Iterative Policy Evaluation: a Small Gridworld Example

- Let consider gridworld environment with two terminal states, where
  - $\gamma = 1$
  - $r(s,a) = -1 \quad \forall s, a$



we want to evaluate this policy



terminal state

$V_0 = 0$

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

V_1

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

V_2

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

V_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

V_{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

V_∞

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

(Here we're not using the in-place update, but the *)

Policy Improvement

Policy Improvement

- Do you remember how to derive optimal policy from optimal value functions?

$$\pi^*(s) = \arg\max_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s') \right\} = \arg\max_a Q^*(s,a)$$

- What happens if we act greedy with respect to non optimal value function?

$$\pi'(s) = \arg\max_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \right\} = \arg\max_a Q_\pi(s,a), \quad \forall s \in \mathcal{S}$$

- Is π' different from π ?
  - If not, it means that π is already the optimal policy π^* (as it satisfies the Bellman Optimality equations)
  - Otherwise, is π' better or as good as π?

Yes, they'll be equal only when $\pi = \pi' = \pi^*$ (optimal)

If $\pi'(s) \neq \pi(s)$ then: $Q_\pi(s, \pi(s)) \leq Q_\pi(s, \pi'(s))$

Policy Improvement Theorem

- For any pair deterministic policies π' and π such that:

$$Q_\pi(s, \pi'(s)) \geq Q_\pi(s, \pi(s)), \quad \forall s \in \mathcal{S}$$

then π' is better or as good as π

$$\pi' \geq \pi$$

  - If $\exists s \in \mathcal{S}$ s.t. $Q_\pi(s, \pi'(s)) > Q_\pi(s, \pi(s))$ then $\pi' > \pi$

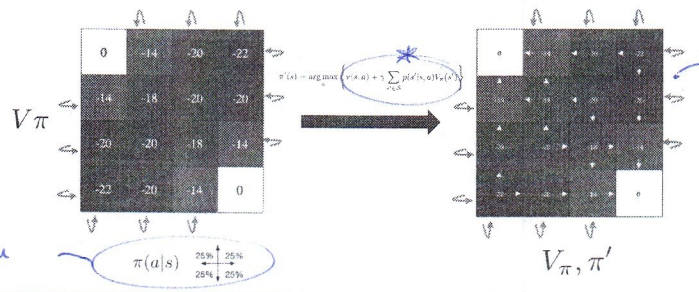
- Proof

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

$Q_\pi(s, \pi(s)) =$

Policy Improvement: a Small Gridworld Example

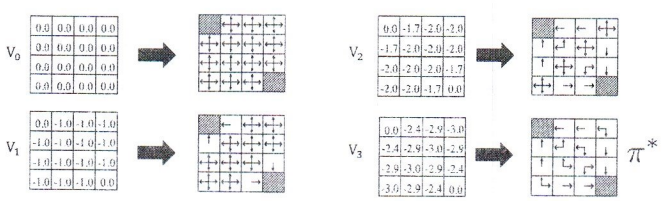
- Let's go back to the value function we found iteratively for a random policy in the Small Gridworld example



random policy

The arrows represent the new actions that maximize * according to V_π

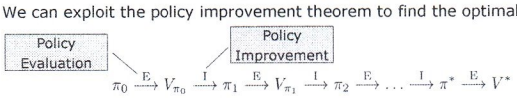
Policy Improvement: a Small Gridworld Example



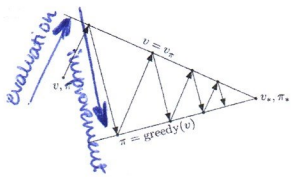
Policy Iteration

Policy Iteration

- We can exploit the policy improvement theorem to find the optimal policy:



We start from a policy π_0 (can be anything, even random), we evaluate it and we get its value function V_{π_0}. Then we apply the policy improvement step and we get a new policy. For the policy improvement theorem we know that $\pi_1 \geq \pi_0$ (better). We do the evaluation of π_1, we get V_{π_1}, we do the improvement,.. Until we reach the optimal.



Policy Iteration (2)

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

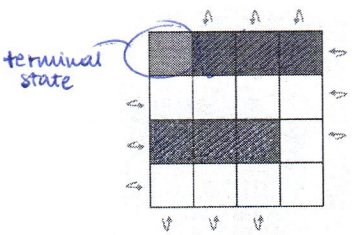
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

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

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   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

Policy Iteration: Example

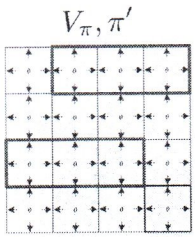
Goal: we want to reach the terminal state with the smaller price to pay



for this reason it's convenient to avoid the blue states

$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$

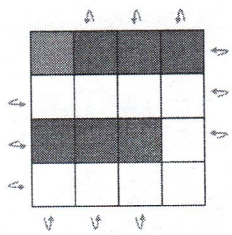


value function V_π initialized to 0's and initial policy completely random

Evaluation
Improvement

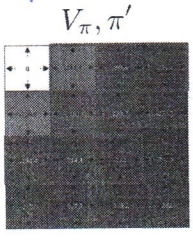
Policy Iteration: Example

We perform: evaluation (of the completely random policy)



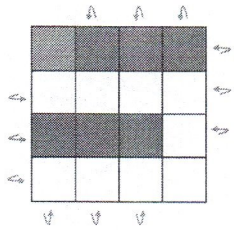
$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



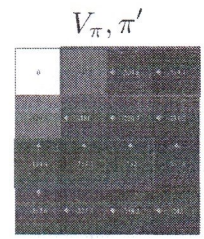
Evaluation
Improvement

Policy Iteration: Example



$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



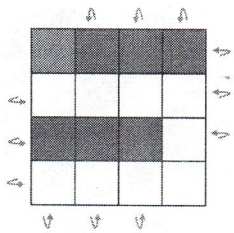
Evaluation

Improvement

(using the knowledge discovered evaluating the random policy)

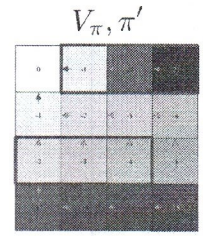
The arrows (choices / actions) are now modified: we created a new policy

Policy Iteration: Example



$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$

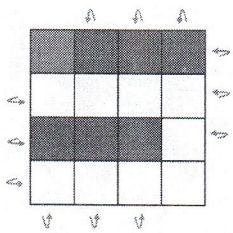


Evaluation

Improvement

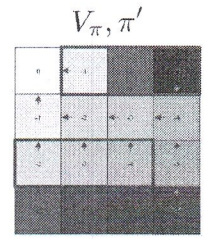
The new policy is already more meaningful

Policy Iteration: Example



$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$

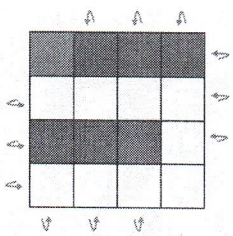


Evaluation

Improvement

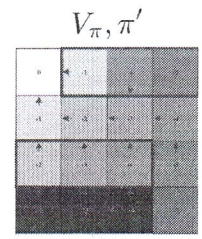
we create a 3rd policy

Policy Iteration: Example



$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

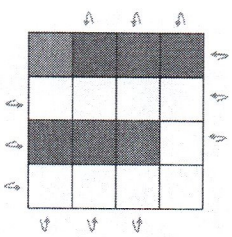
$\gamma = 1$



Evaluation

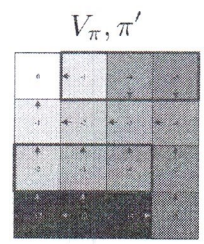
Improvement

Policy Iteration: Example



$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

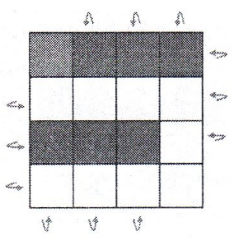
$\gamma = 1$



Evaluation

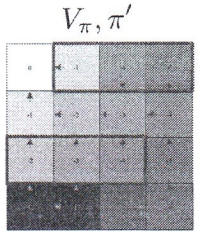
Improvement

Policy Iteration: Example



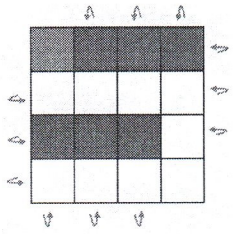
$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



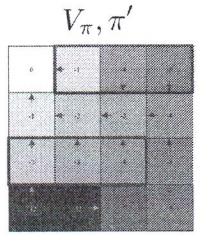
Evaluation
Improvement

Policy Iteration: Example



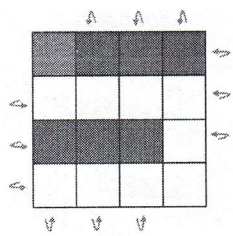
$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



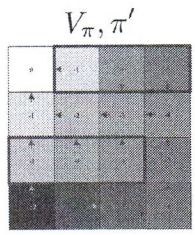
Evaluation
Improvement

Policy Iteration: Example



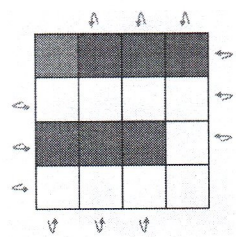
$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



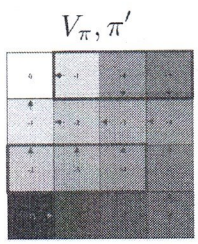
Evaluation
Improvement

Policy Iteration: Example



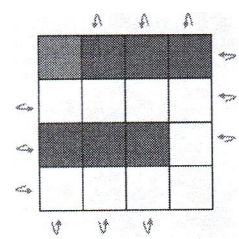
$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



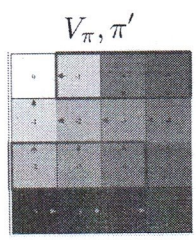
Evaluation
Improvement

Policy Iteration: Example



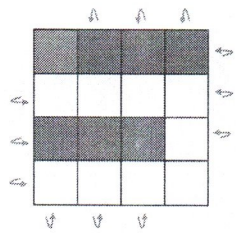
$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



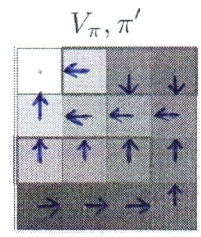
Evaluation
Improvement

Policy Iteration: Example



$$R_t = \begin{cases} -10 & \text{in blue states} \\ -1 & \text{in other states} \end{cases}$$

$\gamma = 1$



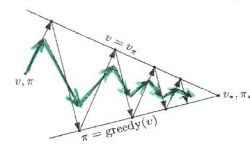
Optimal policy and value function

Reasonably it tries to avoid the blue zones, it, however, we end up in a blue zone we go out of it immediately

Generalized Policy Iteration

Generalized Policy Iteration

- Policy iteration alternates complete policy evaluation and improvement up to the convergence:



- Policy iteration framework allows also to find the optimal policy interleaving partial evaluation and improvement steps
- In particular, Value Iteration is one of the most popular GPI method

Value Iteration

- In the policy evaluation step, only a single sweep of updates is performed:

improvement: $\pi'(s) = \arg\max_a \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_\pi(s') \right\}, \forall s \in \mathcal{S}$

evaluation: $V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi'(a|s) \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \right), \forall s \in \mathcal{S}$

we combine the one-step improvement and the one-step evaluation: we do them together at once

- Combining them, we simply need to iterate the update of the value function using the Bellman optimality equation:

$$V_{k+1}(s) \leftarrow \max_a \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_k(s') \right], \forall s \in \mathcal{S}$$

This does not try to compute explicitly the policy, only the value function. At the very end we still need to compute the optimal policy: once we have the optimal value function we derive it (through argmax)

- It can be proved that $\lim_{k \to \infty} V_k = V^*$

Value Iteration (2)

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 $V(\text{terminal}) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

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

Efficiency of DP

Asynchronous Dynamic Programming

- All the DP methods described so far require exhaustive sweeps of the entire state set.
- Asynchronous DP does not use sweeps:
  - Pick a state at random
  - Apply the appropriate backup (= update)
  - Repeat until convergence criterion is met
- Can you select states to backup intelligently?
  - An agent's experience can act as a guide.

Efficiency of DP

- To find an optimal policy is **polynomial** in the number of states and actions
  - Value Iteration: $O(|S|^2|A|)$ (instead of the brute force $O(|A|^{|S|})$)
  - Policy Iteration: iterative evaluation $O\left(\frac{|S|^2 \log(1/\epsilon)}{\log(1/\gamma)}\right)$, improvement $O\left(\frac{|A|}{1-\gamma}\log\left(\frac{|S|}{1-\gamma}\right)\right)$
- Unfortunately, the number of states is often astronomical, e.g., often growing exponentially with the number of state variables (**curse of dimensionality**)
- In practice, classical DP can be applied to problems with a few millions of states
- Asynchronous DP can be applied to larger problems, and is appropriate for parallel computation
  - But it is **easy** to come up with MDPs for which DP methods are not practical
- **Linear programming** approaches can be also used instead of DP but they do not typically scale well on larger problems