TÉCNICO LISBOA

Planning, Learning and Decision Making

Lecture 8. Markov decision problems (conc.)

Markov decision problem

- **Model** for sequential decision problem
- Described by:
  - State space, $\mathscr{X}$
  - Action space, $\mathscr{A}$
  - Transition probabilities, $\{\mathbf{P}_a, a \in \mathscr{A}\}$
  - Immediate cost function, $\mathbf{c}$
  - Discount factor, γ

Cost-to-go function

- Cost-to-go function J^{π}:
  - Fix a policy, π
  - Start the agent in state x
  - Let the agent go
  - Keep track of all costs to pay

Cost-to-go function

- How much will the agent pay?
  - Depends on the policy π
  - Depends on the initial state x

$$J^{\pi}(x) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t \mathrm{c}_t \;\middle|\; \mathrm{x}_0 = x\right]$$

Policy

Discounted cost-to-go

Initial state

Cost-to-go function

- J^π maps each state in $\mathcal{X}$ to a real value (the discounted cost-to-go)
- The cost-to-go function verifies

$$J^\pi(x) = \sum_{a \in \mathcal{A}} \pi(a \mid x) \underbrace{\left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^\pi(y) \right]}_{Q^\pi(x,a)}$$

Optimal cost-to-go function

- The cost-to-go function for the optimal policy
- The optimal cost-to-go function verifies

$$J^*(x) = \min_{a \in \mathcal{A}} \underbrace{\left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^*(y) \right]}_{Q^*(x,a)}$$

Q-functions

- Represent the (average) cost-to-go for fixed policy, given initial state **and action**
- Q^π and Q^* map each state-action pair in $\mathcal{X} \times \mathcal{A}$ to a real value (the discounted cost-to-go)

Why should we care?

- Q-functions are closely related with cost-to-go functions:

$$J^\pi(x) = \sum_{a \in \mathcal{A}} \pi(a \mid x) \underbrace{\left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^\pi(y) \right]}_{Q^\pi(x,a)}$$

$$J^*(x) = \min_{a \in \mathcal{A}} \underbrace{\left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^*(y) \right]}_{Q^*(x,a)}$$

Why should we care?

- Q-functions are closely related with cost-to-go functions:

$$J^{\pi}(x) = \sum_{a \in \mathcal{A}} \pi(a \mid x) Q^{\pi}(x, a)$$

$$J^{*}(x) = \min_{a \in \mathcal{A}} Q^{*}(x, a)$$

Why should we care?

- The optimal Q-function is closely related with the optimal policy:

$$\pi^*(x) = \underset{a \in \mathcal{A}}{\operatorname{argmin}} \left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^*(y) \right]$$

Why should we care?

- The optimal Q-function is closely related with the optimal policy:

$$\pi^*(x) = \operatorname*{argmin}_{a \in \mathcal{A}} Q^*(x, a) \qquad (\dagger)$$

TÉCNICO LISBOA

Bottom line…

- Q-functions make our life easier
- But can we compute them directly?

Yes.

- Recall that

$$J^{\pi}(x) = \sum_{a \in \mathcal{A}} \pi(a \mid x) \left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^{\pi}(y) \right]$$

Yes.

- Recall that

$$Q^\pi(x,a) = c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^\pi(y)$$

and

$$J^\pi(x) = \sum_{a \in \mathcal{A}} \pi(a \mid x) Q^\pi(x,a)$$

$$Q^\pi(x,a) = c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) \sum_{a' \in \mathcal{A}} \pi(a' \mid y) Q^\pi(y,a')$$

And yes.

- Recall that

$$J^*(x) = \min_{a \in \mathcal{A}} \boxed{c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^*(y)}$$

And yes.

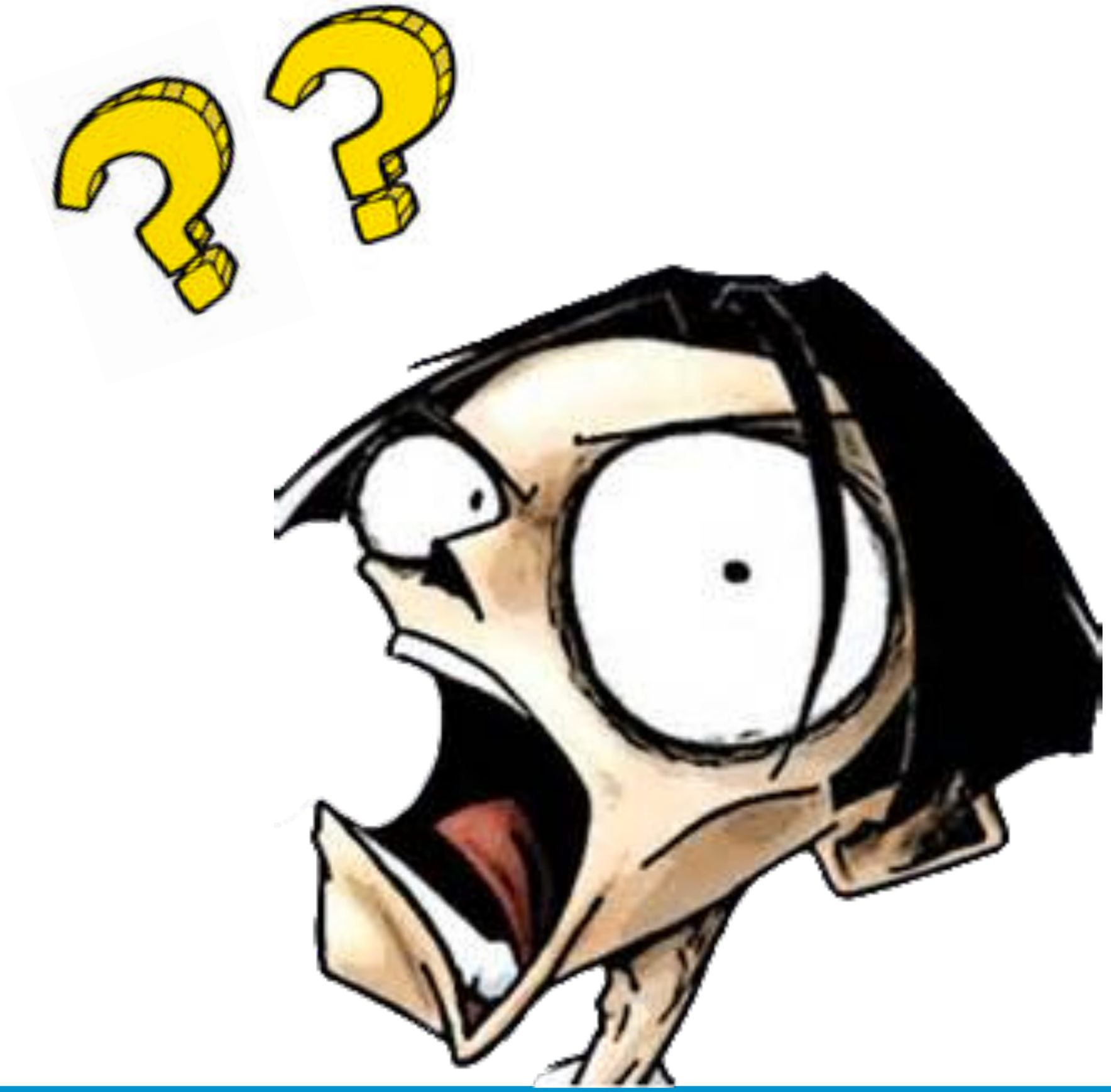
- Recall that

$$Q^*(x,a) = c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^*(y)$$

and

$$J^*(x) = \min_{a \in \mathcal{A}} Q^*(x,a)$$

$$Q^*(x,a) = c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) \min_{a' \in \mathcal{A}} Q^*(y,a')$$



DON'T PANIC

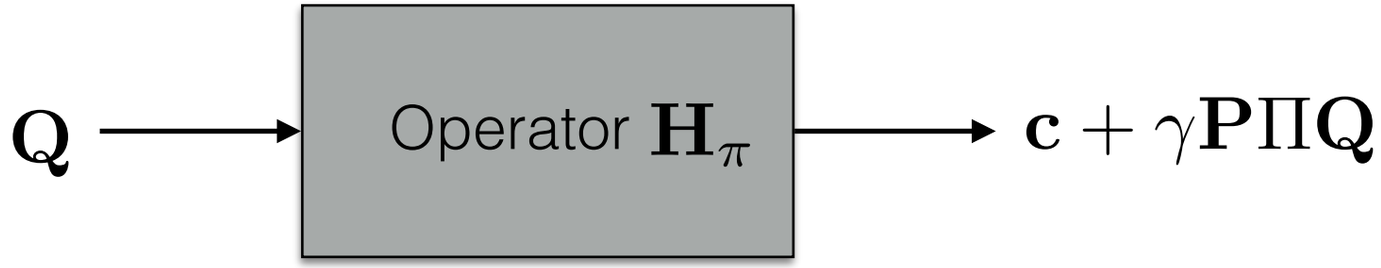
Just two recursive equations…

$$Q^{\pi}(x,a) = c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) \sum_{a' \in \mathcal{A}} \pi(a' \mid y) Q^{\pi}(y,a')$$

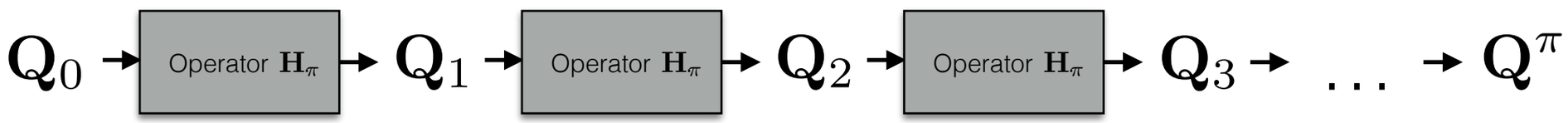
$$Q^{*}(x,a) = c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) \min_{a' \in \mathcal{A}} Q^{*}(y,a')$$

… that we can iterate upon

$$Q^\pi(x,a) = \underbrace{c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) \sum_{a' \in \mathcal{A}} \pi(a' \mid y) Q^\pi(y,a')}_{\mathbf{H}_\pi}$$

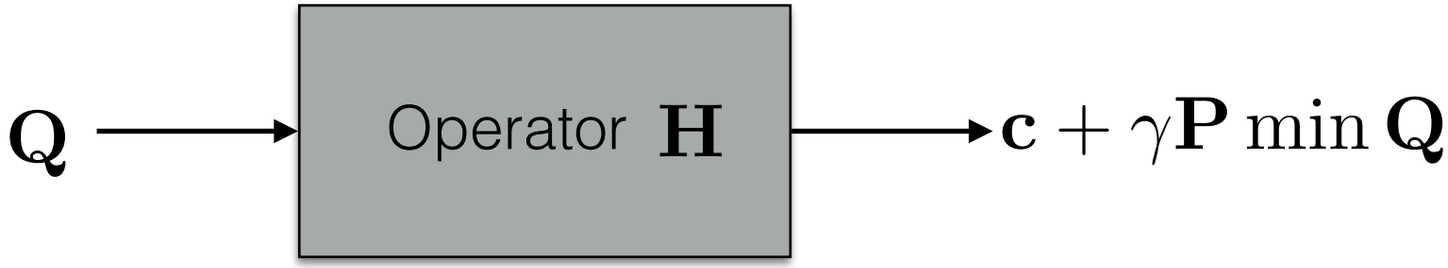


- To get:

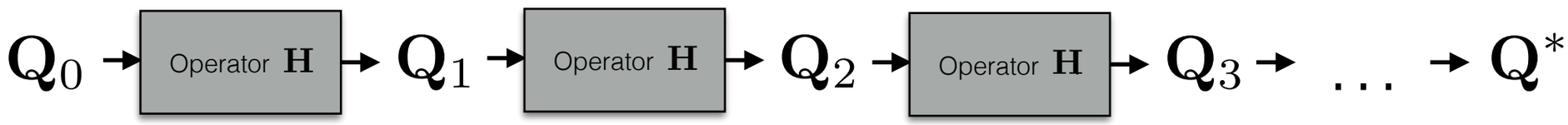


… that we can iterate upon

$$Q^*(x,a) = \underbrace{c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) \min_{a' \in \mathcal{A}} Q^*(y,a')}_{\mathbf{H}}$$



- To get:



Value Iteration, 3.0

Require: MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \{\boldsymbol{P}_a\}, c, \gamma)$; tolerance $\varepsilon > 0$;

Initialize $k = 0$, $Q^{(0)} \equiv 0$

repeat

  $Q^{(k+1)} \leftarrow \mathsf{H} Q^{(k)}$

  $k \leftarrow k + 1$

until $\left\| Q^{(k-1)} - Q^{(k)} \right\| \quad < \varepsilon.$

Compute π^* using (†)

return π^*

Policy iteration

Value iteration

- Value iteration methods are guaranteed to converge asymptotically

Can we do better?

Our goal

- Our goal is to compute the optimal policy
- How many policies are there?

$$|\mathcal{A}|^{|\mathcal{X}|}$$

Finite number!

- If we search all, we'll finish in a finite number of iterations!

Policy iteration

- Build a sequence of policies,

$$\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(k)}, \pi^{(k+1)}, \ldots$$

  where each one is better than the previous

Greedy policy

- Given a cost-to-go function *J*, the greedy policy w.r.t. *J* is

$$\pi_g(x) = \underset{a \in \mathcal{A}}{\operatorname{argmin}} \left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}(y \mid x, a) J(y) \right]$$

- … for example, the optimal policy is greedy w.r.t. *J**

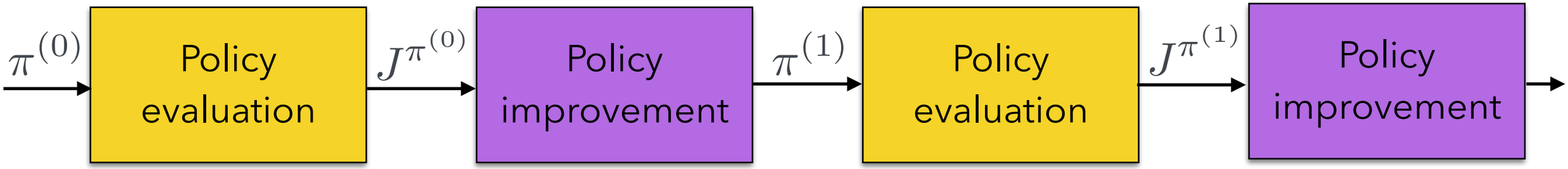
Improvement result

- Given a policy π
- … given the cost-to-go function J^{π},
- … then

$$J^{\pi_g}(x) \leq J^{\pi}(x)$$

Policy π_g is better than π

Policy iteration



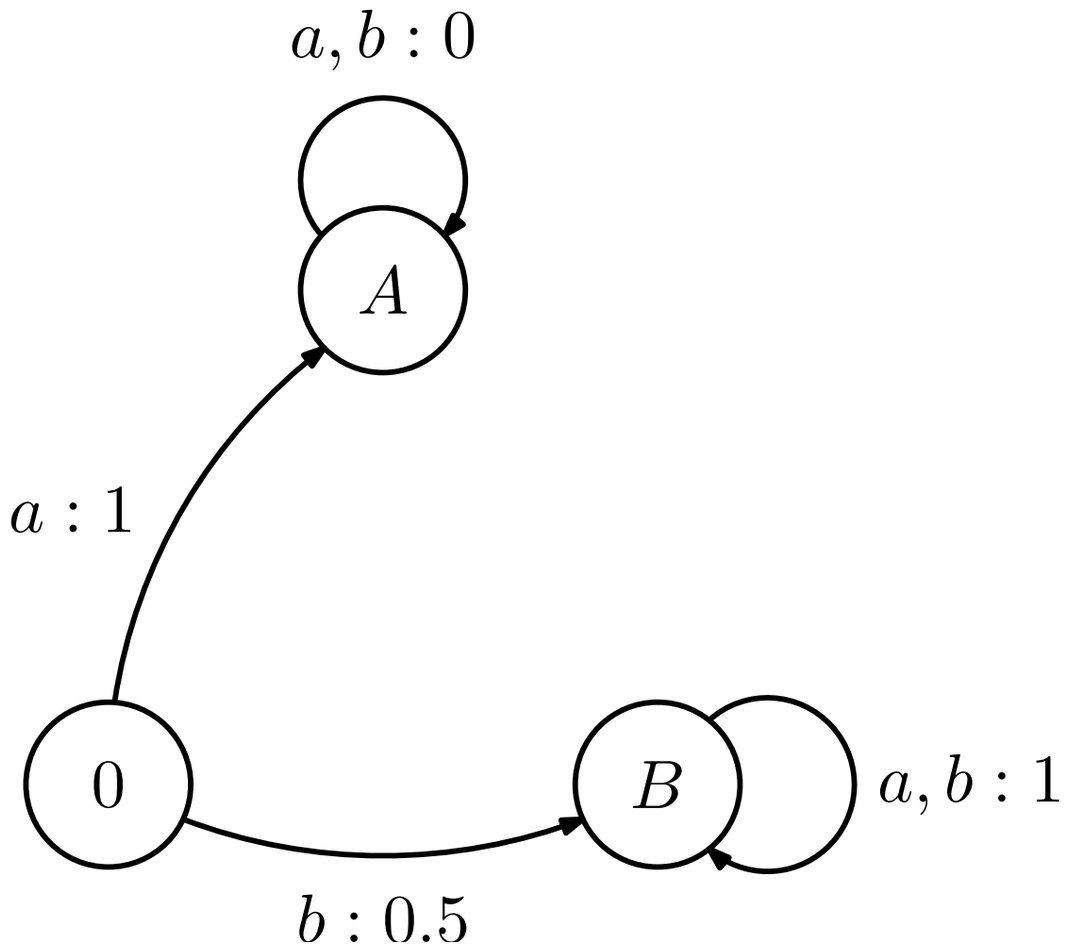
Policy iteration

Require: MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \{\boldsymbol{P}_a\}, c, \gamma)$

1: Initialize $k = 0$, $\pi^{(0)}$ randomly
2: **repeat**
3: $\quad \boldsymbol{J} \leftarrow (\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^{(k)}})^{-1} \boldsymbol{c}_{\pi^{(k)}}$ Policy evaluation
4: $\quad \pi^{(k+1)} \leftarrow \pi_g^J$ Policy improvement
5: $\quad k \leftarrow k + 1$
6: **until** $\pi^{(k-1)} = \pi^{(k)}$
7: **return** $\pi^{(k)}$

Example

- Let us compute the optimal policy using PI
- Start with $\pi^{(0)}$ as the uniform policy



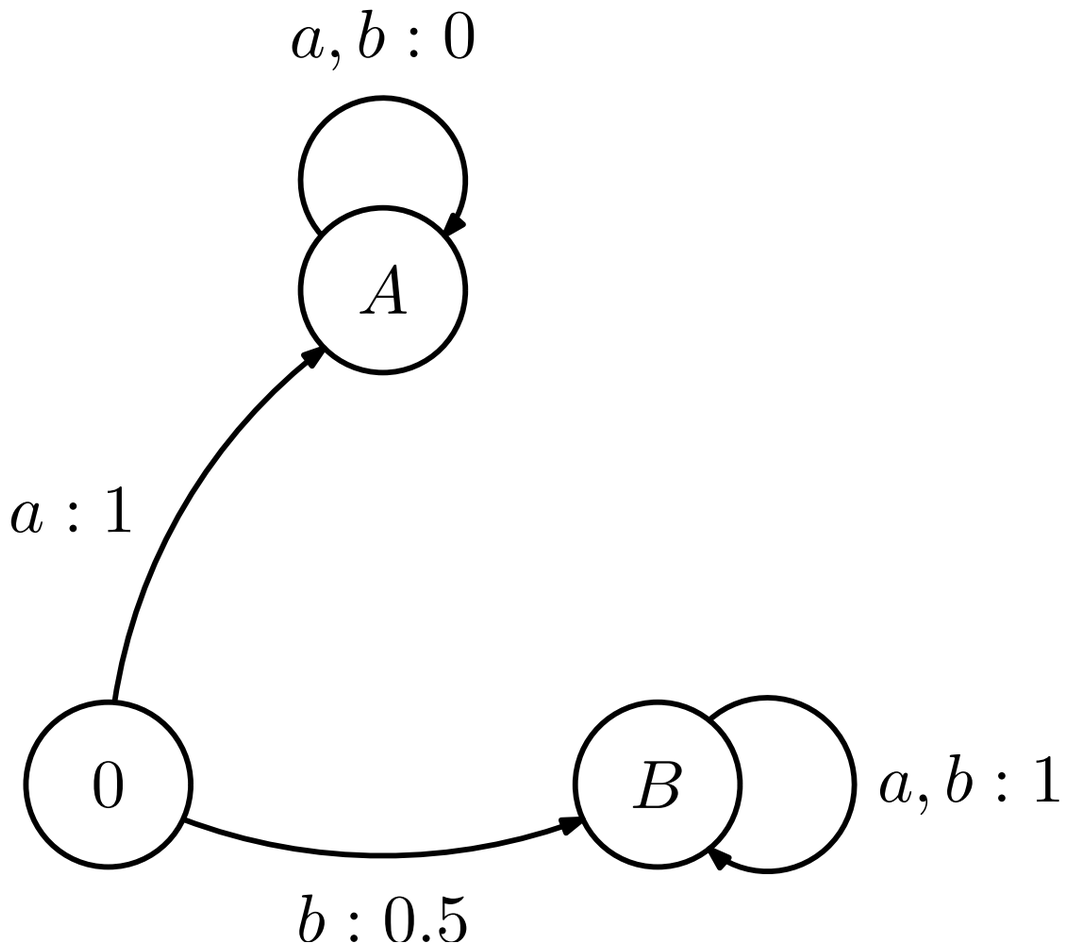
Example

- Step 1 - Evaluate π:

$$\mathbf{c}_\pi = \begin{bmatrix} 0.75 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{P}_\pi = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{J}^\pi = \begin{bmatrix} 50.25 \\ 0 \\ 100 \end{bmatrix}$$

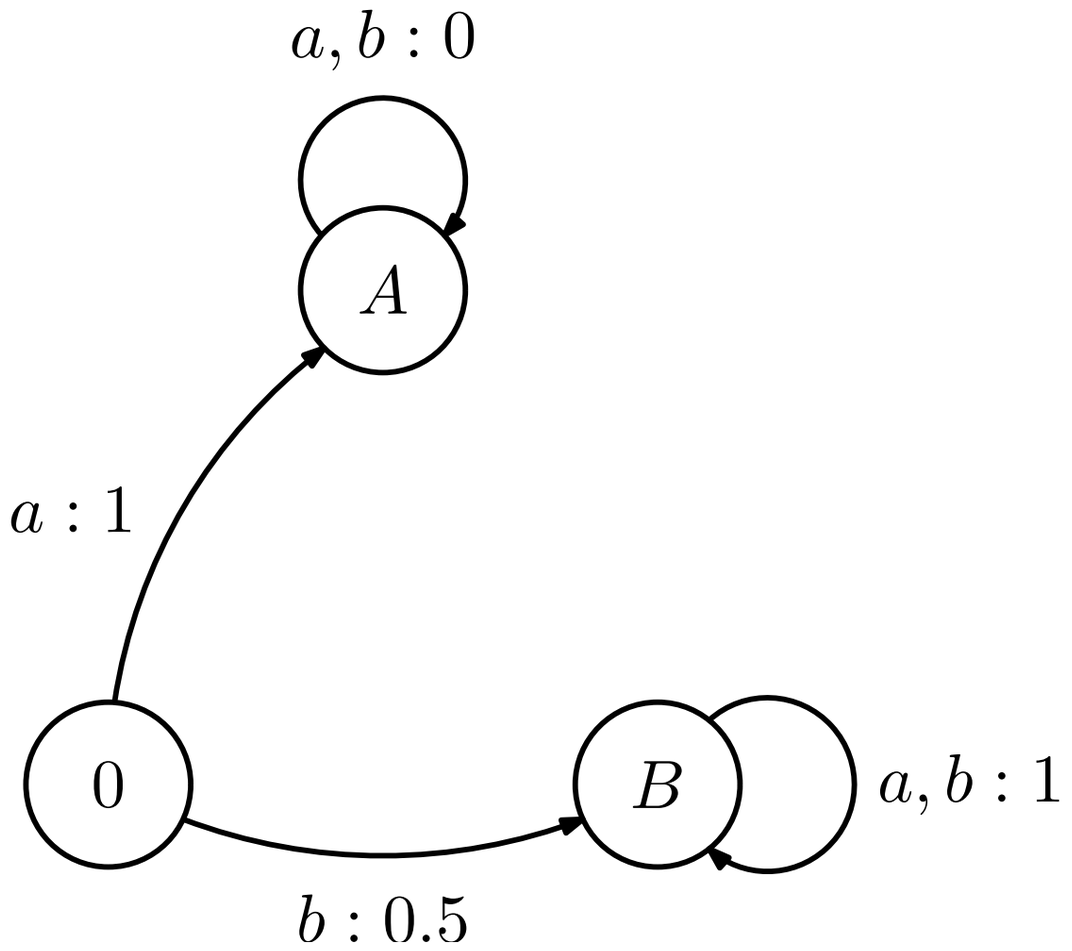


Example

- Step 2 - Improve π:

$$\mathbf{Q}^{\pi} = \begin{bmatrix} 1 & 99.5 \\ 0 & 0 \\ 100 & 100 \end{bmatrix}$$

$$\pi^{(1)} = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$



… and we're done!

Example

fmelo — python — 80×30

```
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>>
>>> ca = np.array([[1],[0], [1]])
>>> cb = np.array([[0.5],[0], [1]])
>>> Pa = np.array([[0, 1, 0], [0, 1, 0], [0, 0, 1]])
>>> Pb = np.array([[0, 0, 1], [0, 1, 0], [0, 0, 1]])
>>> gamma = 0.99
>>>
>>> J = np.zeros((3, 1))
>>> err = 1
>>> i = 0
>>>
>>> while err > 1e-8:
...     Qa = ca + gamma * Pa.dot(J)
...     Qb = cb + gamma * Pb.dot(J)
...     Jnew = np.min((Qa, Qb), axis=0)
...     err = np.linalg.norm(Jnew - J)
...     i += 1
...     J = Jnew
...
>>> print(J)
 [[  1.        ]
  [  0.        ]
  [ 99.99999901]]
>>> print(i)
1834
>>>
```

Value iteration cycle

1834 iterations!

Example

fmelo — python — 108×45

```
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>>
... ca = np.array([[1], [0], [1]])
>>> cb = np.array([[0.5], [0], [1]])
>>> Pa = np.array([[0, 1, 0], [0, 1, 0], [0, 0, 1]])
>>> Pb = np.array([[0, 0, 1], [0, 1, 0], [0, 0, 1]])
>>> gamma = 0.99
>>>
... pi = np.ones((3, 2)) / 2 # Initial policy
>>> quit = False
>>> i = 0
>>>
>>> print('pi0:', pi)
pi0: [[ 0.5  0.5]
 [ 0.5  0.5]
 [ 0.5  0.5]]
>>>
... while not quit:
...     cpi = np.diag(pi[:, 0]).dot(ca) + np.diag(pi[:, 1]).dot(cb)
...     Ppi = np.diag(pi[:, 0]).dot(Pa) + np.diag(pi[:, 1]).dot(Pb)
...     J = np.linalg.inv(np.eye(3) - gamma * Ppi).dot(cpi)
...
...     Qa = ca + gamma * Pa.dot(J)
...     Qb = cb + gamma * Pb.dot(J)
...
...     pinew = np.zeros((3, 2))
...     pinew[:, 0, None] = np.isclose(Qa, np.min([Qa, Qb], axis=0), atol=1e-10, rtol=1e-10).astype(int)
...     pinew[:, 1, None] = np.isclose(Qb, np.min([Qa, Qb], axis=0), atol=1e-10, rtol=1e-10).astype(int)
...
...     pinew = pinew / np.sum(pinew, axis=1, keepdims = True)
...
...     quit = (pi == pinew).all()
...     pi = pinew
...     i +=1
...
>>> print(pi)
 [[ 1.   0. ]
  [ 0.5  0.5]
  [ 0.5  0.5]]
>>> print(i)
2
>>>
```

Policy evaluation

Q-function

Improved policy

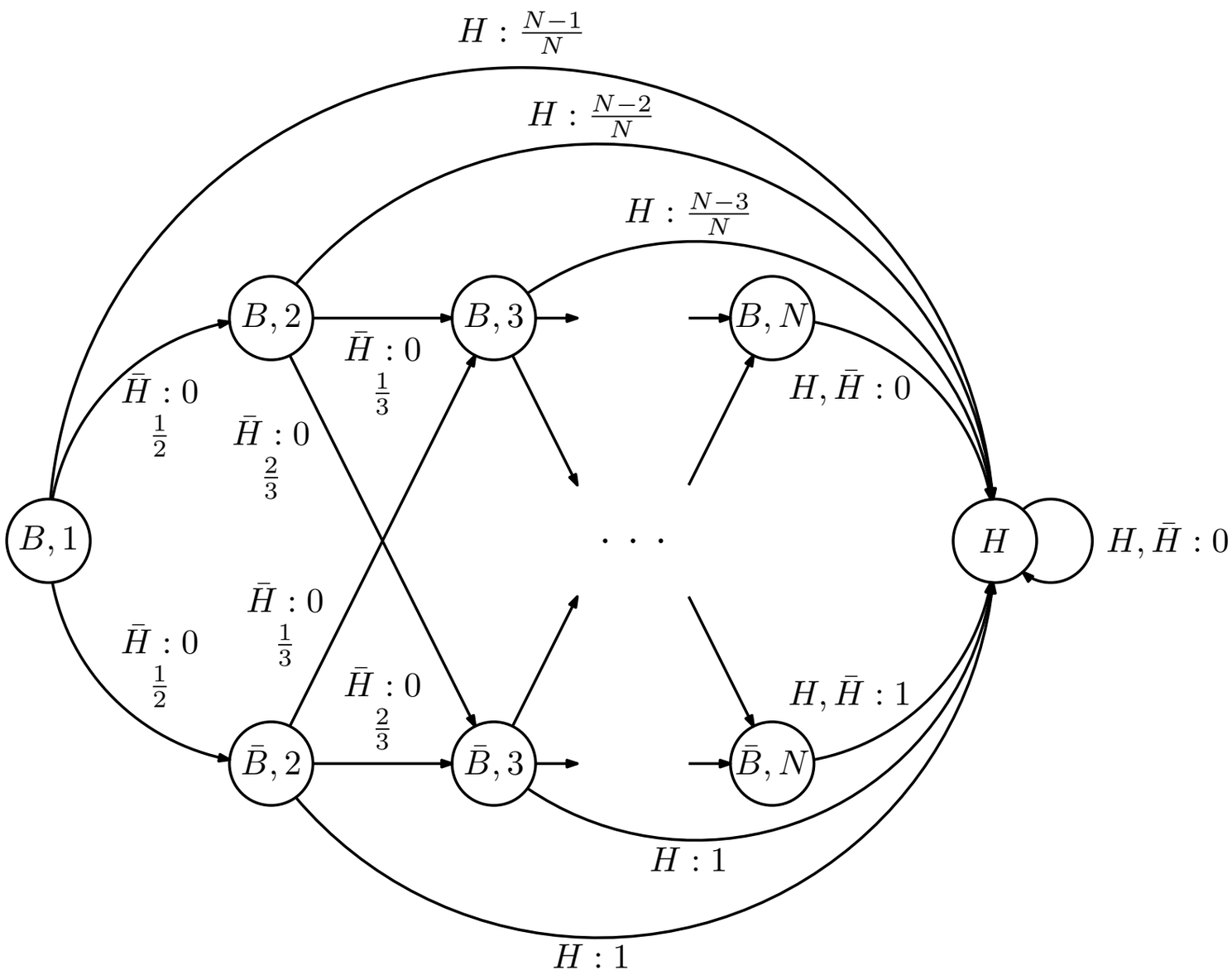
2 iterations!

Example 2

- A company wants to hire a computer engineer
- After initial trial, *N* candidates are selected for interview
- Candidates are interviewed sequentially
- Order of the candidates for interview was selected randomly

Example 2

- The MDP can be summarized by the diagram

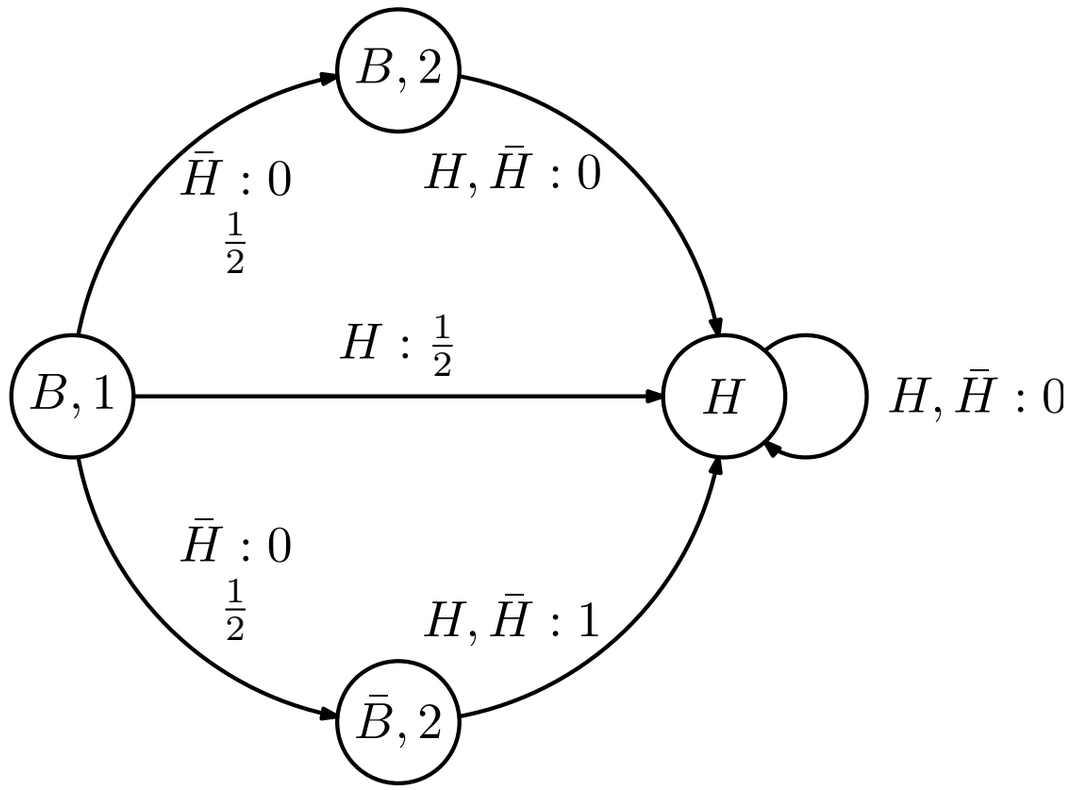


Example 2

For $N = 2$,

- Value iteration concludes in 3 iterations
- Policy iteration concludes in 2 iterations

$$\pi^* = \begin{bmatrix} 0 & 1 \\ 0.5 & 0.5 \\ 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$



Example 2

For $N = 3$,

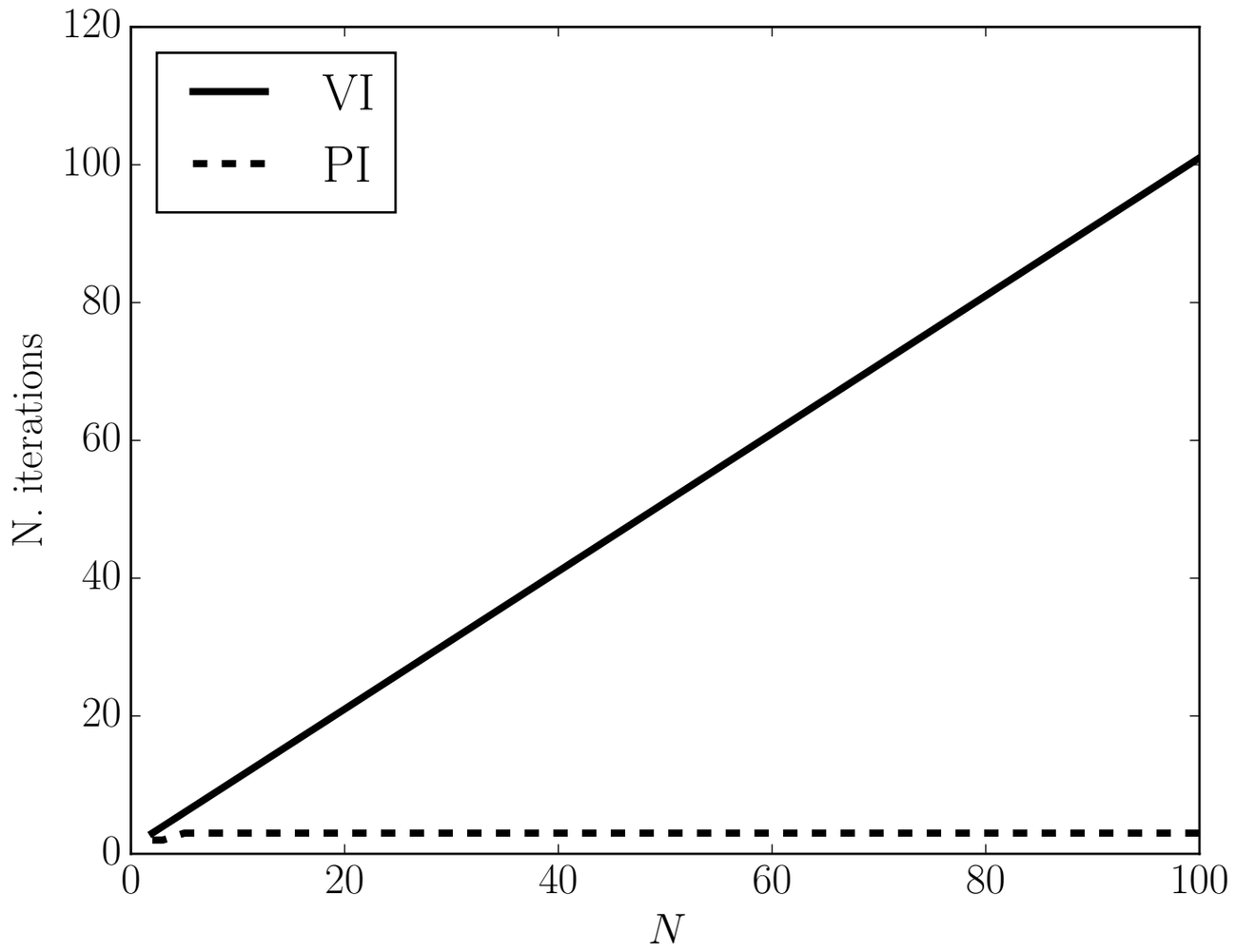
- Value iteration concludes in 4 iterations
- Policy iteration concludes in 2 iterations

$$\pi^* = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0.5 & 0.5 \\ 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

(B, 1)
(B, 2)
(¬B, 2)
(B, 3)
(¬B, 3)
H

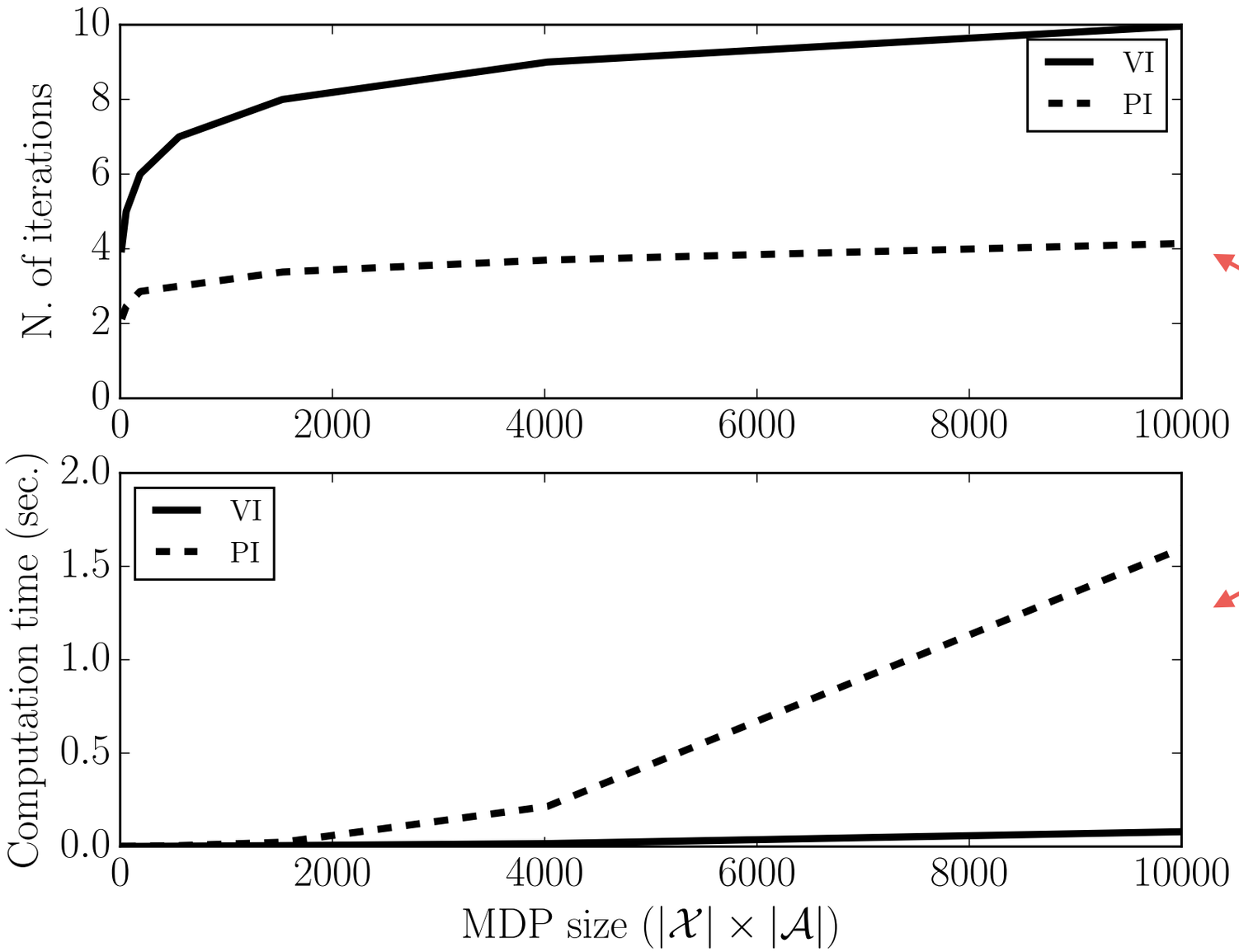
Example 2

As *N* grows:



Careful with this plot!

Another example:



Although VI takes more iterations, it takes less time

TÉCNICO LISBOA

Linear programming

Linear programming

- It is also possible to solve MDPs using linear programming

Linear programming

- A linear program is a **constrained optimization problem**
- Objective function and constraints are **linear**
- General form:

$$\begin{aligned} \max \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \end{aligned}$$

Again the recursion…

- The optimal cost-to-go function verifies

$$J^*(x) = \min_{a \in \mathcal{A}} \left[c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J^*(y) \right]$$

- It is the **largest** function J such that, for all x and all a,

$$J(x) \leq c(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathsf{P}_a(y \mid x) J(y)$$

Again the recursion…

- Putting everything together,

$$\begin{aligned} \max \quad & \boldsymbol{\mu}^\top \mathbf{J} \\ \text{s.t.} \quad & \mathbf{J} \leq \mathbf{c}_a + \gamma \mathbf{P}_a \mathbf{J} \end{aligned}$$

Can be solved with any LP package

Example

fig-prep — python — 80×21

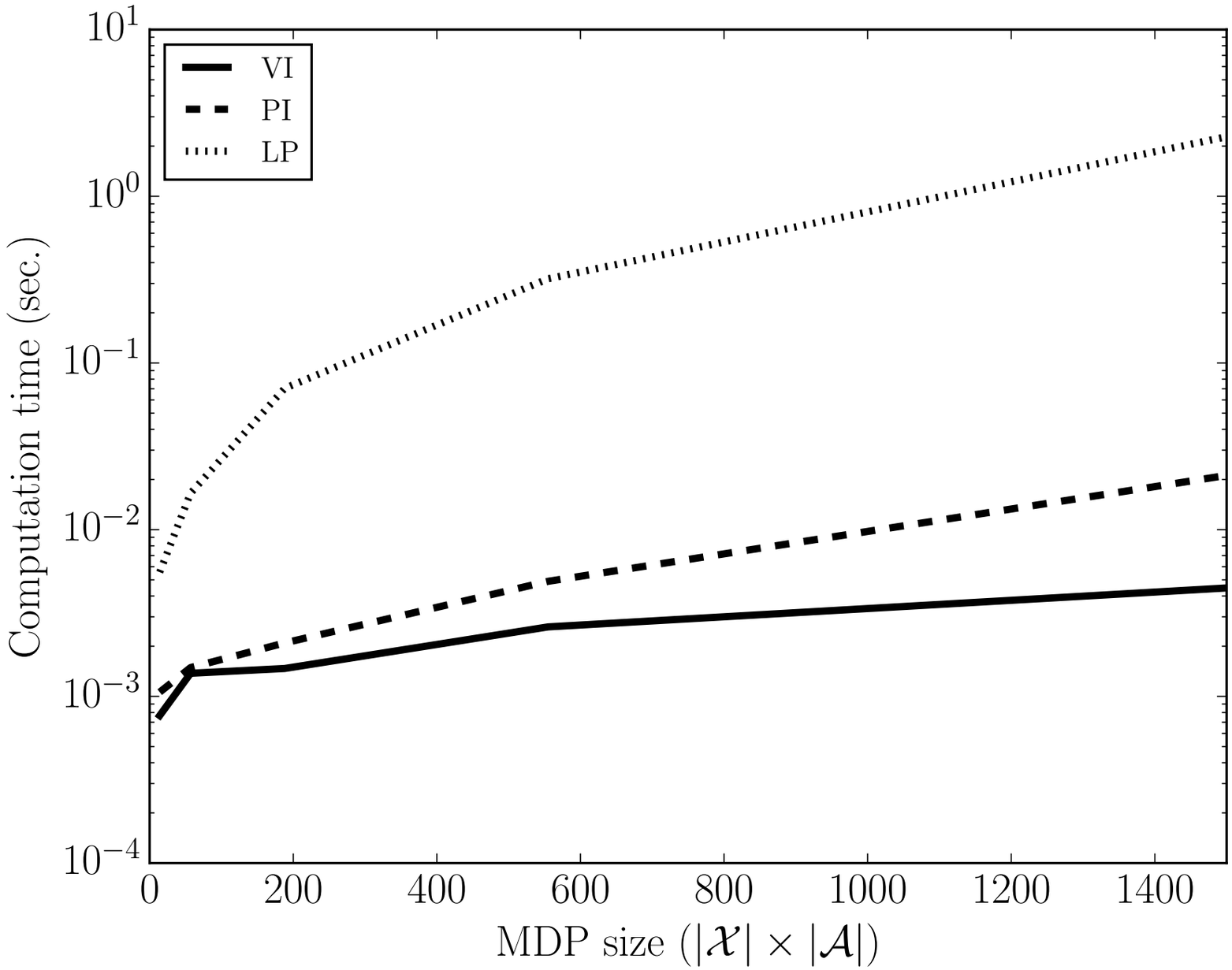
```
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> import scipy.optimize as opt
>>>
>>> ca = np.array([[1], [0], [1]])
>>> cb = np.array([[0.5], [0], [1]])
>>> Pa = np.array([[0, 1, 0], [0, 1, 0], [0, 0, 1]])
>>> Pb = np.array([[0, 0, 1], [0, 1, 0], [0, 0, 1]])
>>> gamma = 0.99
>>>
>>> u = -np.ones(3) / 3
>>>
>>> a_mat = np.concatenate((np.eye(3) - gamma * Pa, np.eye(3) - gamma * Pb))
>>> b = np.concatenate((ca, cb))
>>>
>>> sol = opt.linprog(u, a_mat, b)
>>> print(sol.x)
[  1.   0. 100.]
>>>
```

scipy optimization package

Objective

$\mathbf{Ax} \leq \mathbf{b}$

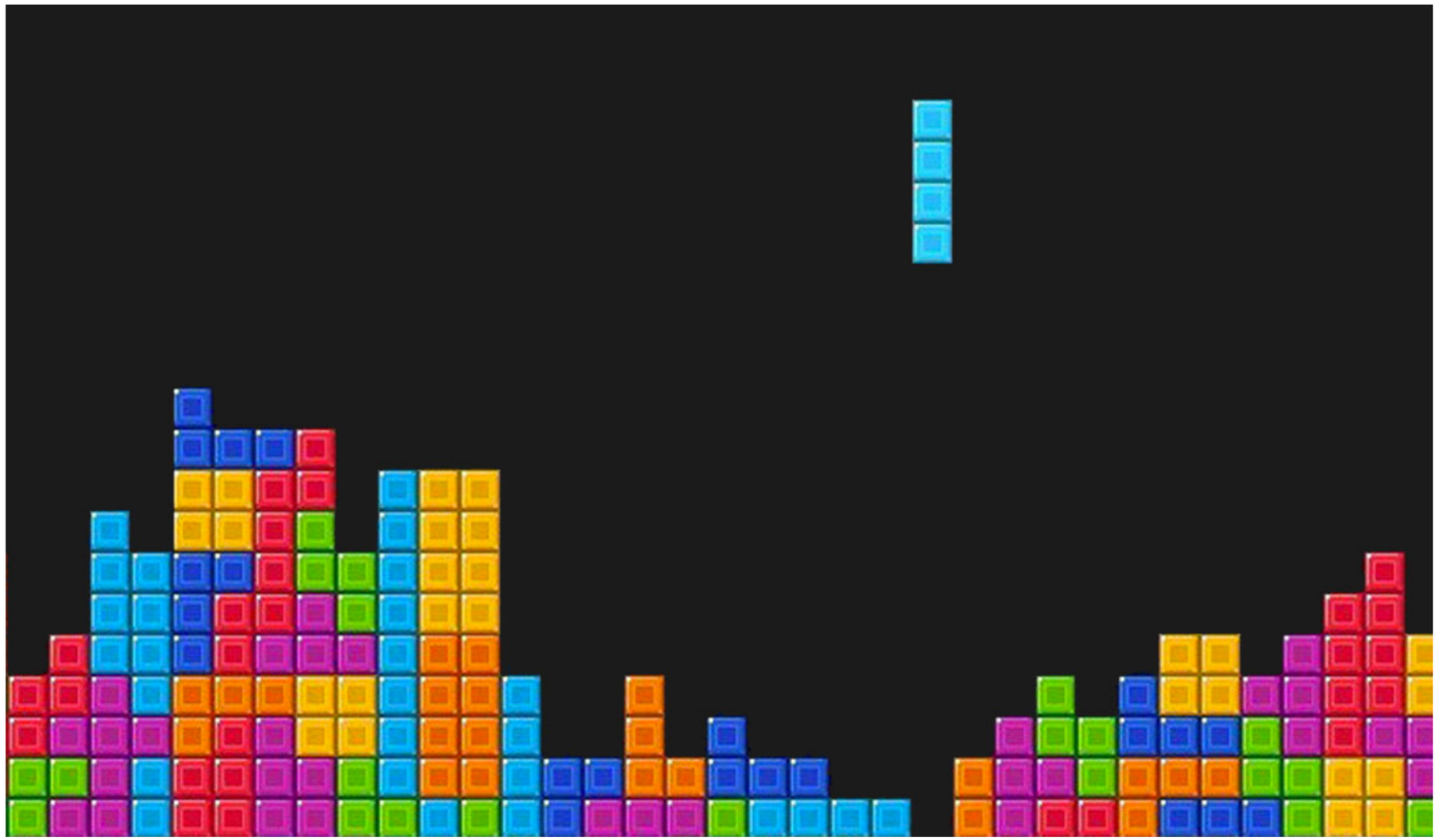
How good is it?



Large problems

The curse of dimensionality

- Larger problems cannot be solved exactly



The curse of dimensionality

- We must resort to some form of approximation
- Instead of allowing arbitrary functions, methods restrict to pre-specified families of functions

- Families provide good representations → methods perform well
- Families provide bad representations → methods perform poorly

Types of approximations

- **State aggregation**
  - State space is partitioned in “chunks”
  - We treat each chunk as a “super-state”

Methods retain performance guarantees

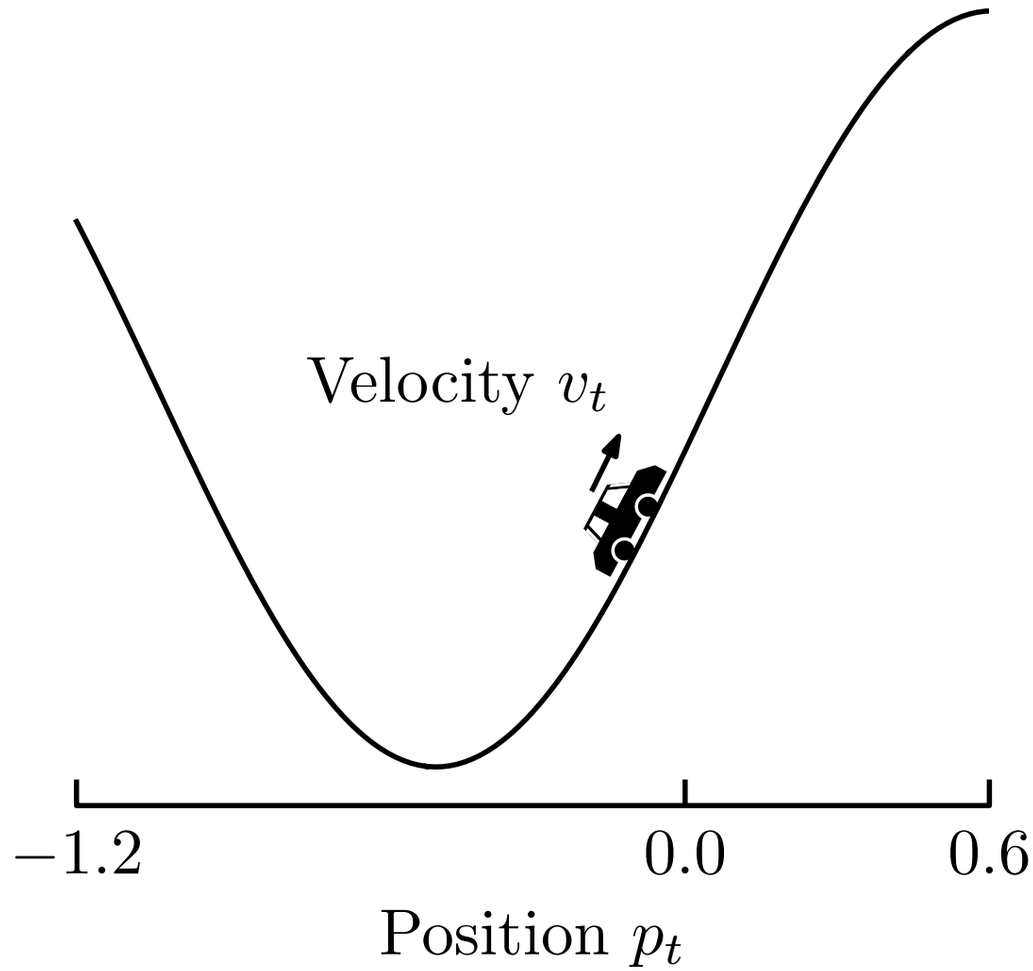
Types of approximations

- **Linear approximations**
  - Each state is described as a vector of “features”
  - *J* and *Q* are represented as combinations of features
  - Methods compute best weights for the combinations

Methods **do not**
retain performance
guarantees

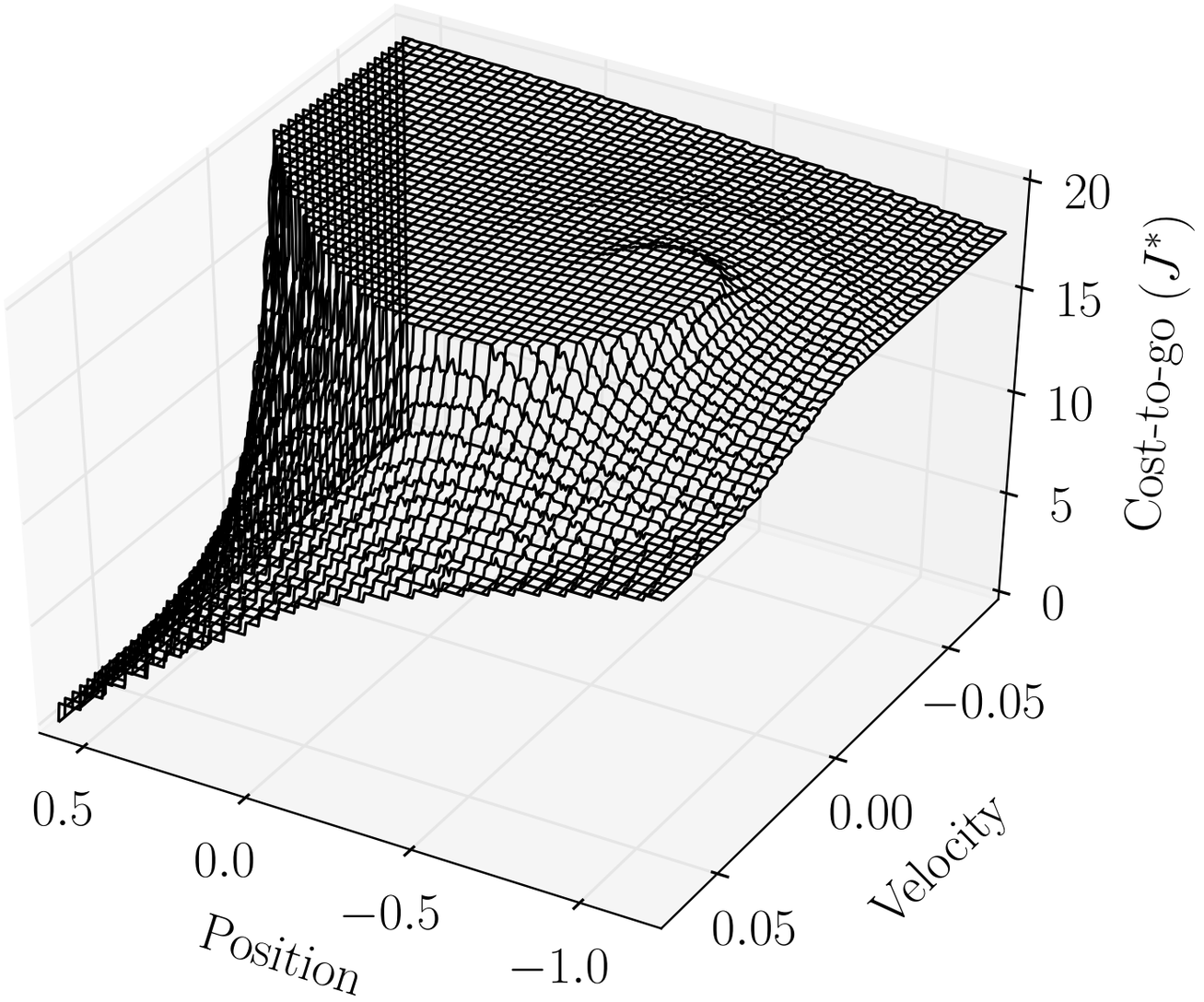
Example: The mountain car

- A car goes up a mountain
- Engine is not strong enough to go all the way up
- Car must run back and forth



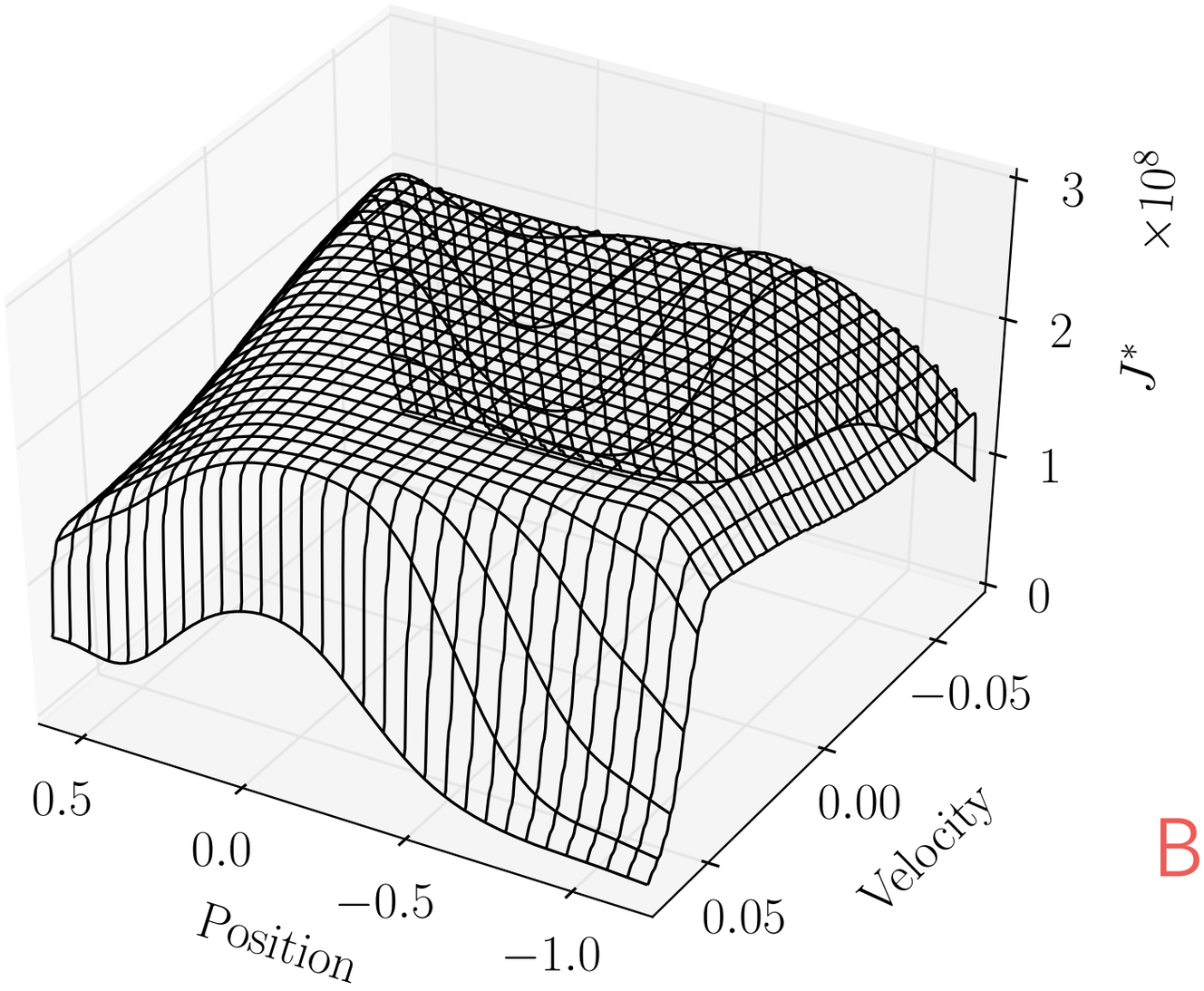
Example: The mountain car

- The actual cost-to-go:



Example: The mountain car

- A naive approximation:



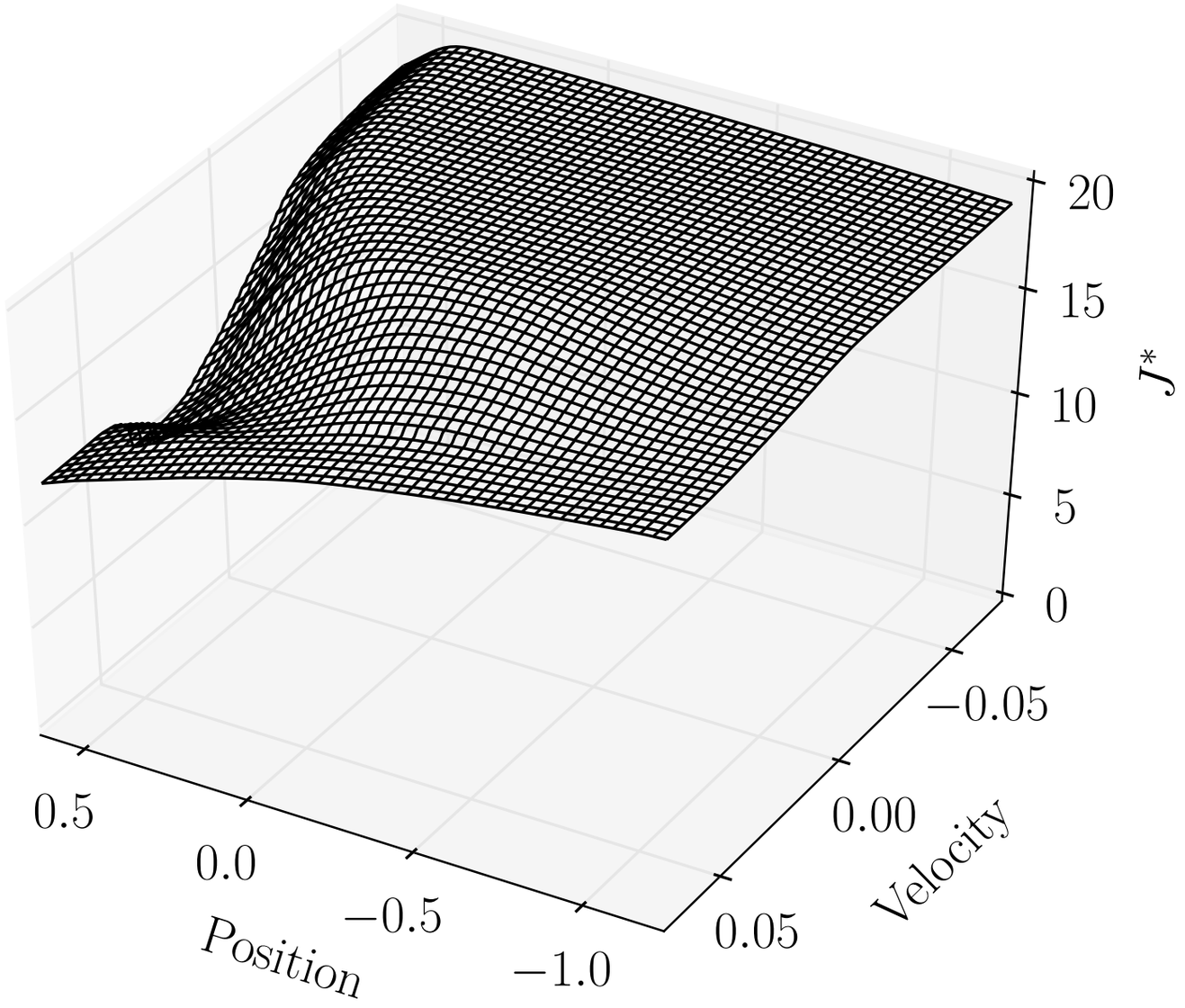
Bad shape and scale
(look at z-axis!)

Types of approximations

- **Averagers**
  - Making sure that the approximation does not **extrapolate**, convergence guarantees can be retained
  - Such approximation architectures are known as **averagers**

Example: The mountain car

- An averager:



Not all the detail,
but the shape is ok