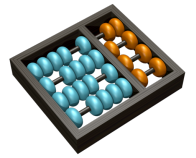
Reinforcement Learning - Lecture 3

Profa. Dra. Esther Luna Colombini
esther@ic.unicamp.br

Prof. Dr. Alexandre Simoes
alexandre.simoes@unesp.br

LaRoCS – Laboratory of Robotics and Cognitive Systems



Advanced Institute for Artificial Intelligence

Summary

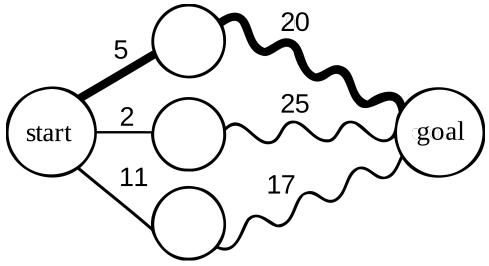
- Introduction
- Policy Evaluation
- Policy Iteration
- Value Iteration
- Extensions to Dynamic Programming
- Contraction Mapping

Disclaimer

- These slides were built upon David Silver's Lecture notes on Reinforcement Learning

What is Dynamic Programming?

- **Dynamic Programming** is a technique for solving complex problems
  - Invented by Richard Bellman in the 1950s to solve optimization problems
  - Main idea
    - It solves the problem by breaking the problem into simple subproblems (overlapping)
    - For each sub-problem, we compute and store the solution (in a table)
    - If the same subproblem occurs, no computation is needed
      - the already computed solution is employed
    - Combine solutions for subproblems
    - Final state of the table contain or will be the solution



Dynamic sequential or temporal component to the problem

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

such as linear programming

Requirements for Dynamic Programming

- **DP** 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

Planning by Dynamic Programming

- Dynamic programming assumes full knowledge of the MDP
- It is used for planning in an MDP
- For prediction:
  - Input:
    - MDP $\langle S, \mathcal{P}, A, \mathcal{R}, \gamma \rangle$ and a policy π
    - MRP $\langle S, \mathcal{P}^{\pi}, R^{\pi}, \gamma \rangle$
  - Output:
    - A value function v_{π}
- For control:
  - Input:
    - MDP $\langle S, \mathcal{P}, A, \mathcal{R}, \gamma \rangle$ and a policy π
  - Output:
    - Optimal value function v_*
    - Optimal policy π_*

Dynamic Programming: Application

- Dynamic programming can be used to solve a variety of problems
  - Scheduling algorithms
  - String algorithms (e.g. sequence alignment)
  - Graph algorithms (e.g. shortest path algorithms)
  - Graphical models (e.g. Viterbi algorithm)
  - Bioinformatics (e.g. lattice models)
  - Computing the global distance between two time series (e.g dynamic time warping algorithm)

Dynamic Programming: Algorithms

- Dynamic programming is a special technique for solving the **Bellman optimality equation**
- We solve a Bellman equation using two powerful algorithms:
  - Value iteration
  - Policy iteration

Value iteration: 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', $v_\pi(s') = v_*(s')$

Value iteration

- In value iteration, we start off with a random value function
- As the random value function might not be an optimal one
  - we look for a new improved value function in iterative fashion until we find the optimal value function
  - with the optimal value function found
    - an optimal policy can be derived

Deterministic Value Iteration

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

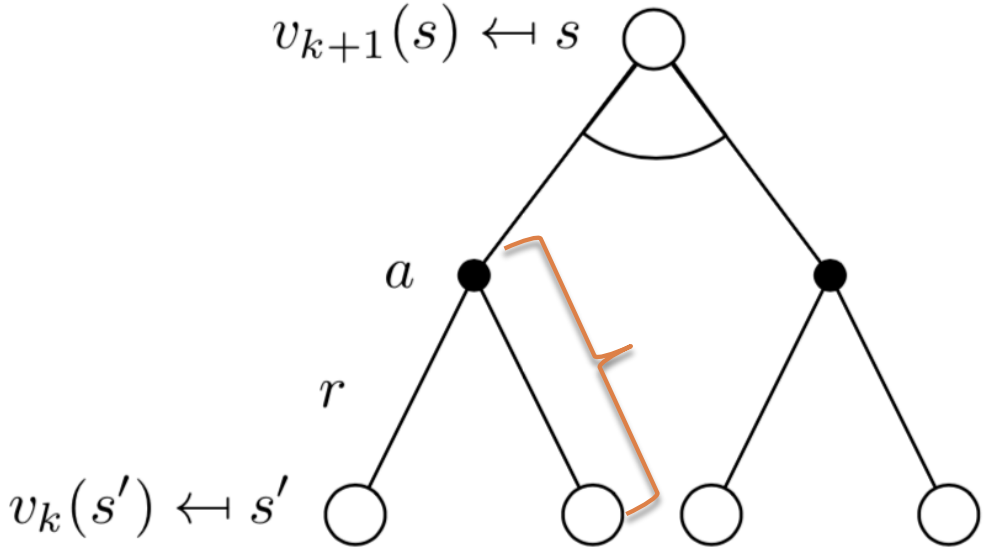
$$v_*(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \right)$$

- 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

Deterministic 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 $s \in S$
  - Update $v_{k+1}(s)$ from $v_k(s')$
- It will converge to v_*
- Unlike policy iteration, there is no explicit policy
- Intermediate value functions may not correspond to any policy

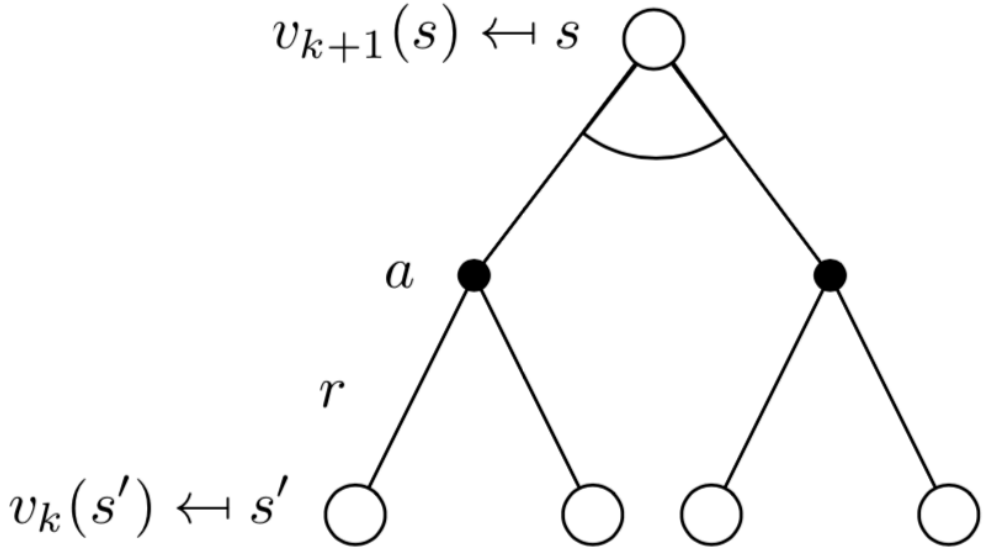
Deterministic Value Iteration



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

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

Deterministic Value Iteration



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

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

Deterministic Value Iteration

R=-1 for each action,, γ=1

g			

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

V_5

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

V_6

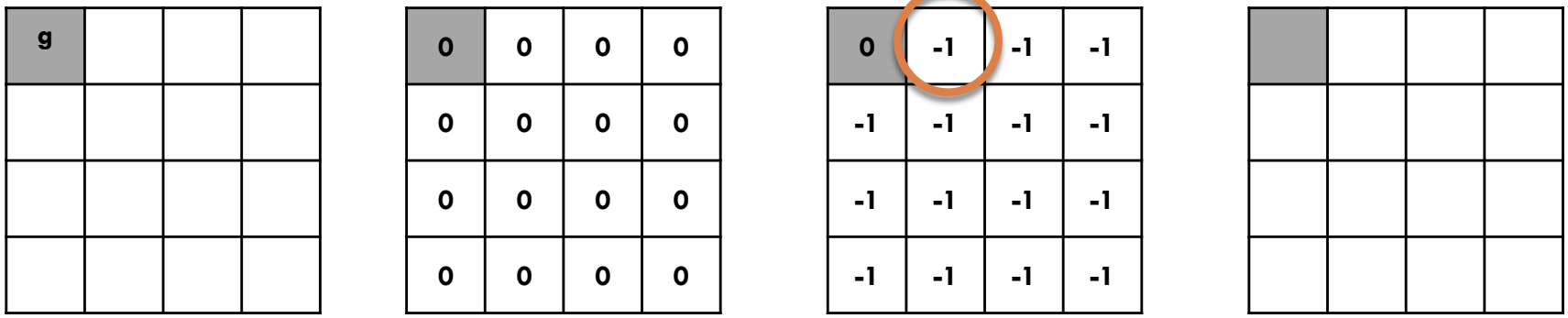
0	-1	-2	-2
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

$V_2(1,2)$= max{-1+0.5*0; -1+0.5*0; -1+0.5*0; -1+0.5*0}



Deterministic Value Iteration

R=-1 for each action,, γ=0.5

$V_3(1,2)= \max\{-1+0.5*0; -1+0.5*-1; -1+0.5*-1; -1+0.5*-1\}$

g			

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		

V_3

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

$V_3(1,3)= \max\{-1+0.5*-1; -1+0.5*-1; -1+0.5*-1; -1+0.5*-1\}$

g			

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	--1,5	

V_3

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

$V_3(1,4) = \max\{-1+0.5*-1; -1+0.5*-1; -1+0.5*-1; -1+0.5*-1\}$

g			

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	--1,5	-1,5

V_3

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

$V_3(2,1)= \max\{-1+0.5*0; -1+0.5*-1; -1+0.5*-1; -1+0.5*-1\}$

g			

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	--1,5	-1,5
-1	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5

V_3

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

g			

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	--1,5	-1,5
-1	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5

V_3

0	-1		

V_4

$V_4(1,2)= \max\{-1+0.5*0; -1+0.5*-1; -1+0.5*-1.5; -1+0.5*-1.5\}$

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

g			

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	--1,5	-1,5
-1	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5

V_3

0	-1	-1.5	

V_4

$V_4(1,3)= \max\{-1+0.5*-1;\ -1+0.5*-1.5;\ -1+0.5*-1.5;\ -1+0.5*-1.5\}$

Deterministic Value Iteration

R=-1 for each action,, γ=0.5

g			

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	--1,5	-1,5
-1	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5
--1,5	--1,5	--1,5	--1,5

V_3

0	-1	-1.5	1.75
-1	-1.5	-1.75	-1.75
-1.5	-1.75	-1.75	-1.75
-1.75	-1.75	-1.75	-1.75

V_4

$V_4(1,4)= \max\{-1+0.5*-1.5; -1+0.5*-1.5; -1+0.5*-1.5; -1+0.5*-1.5\}$
$V_4(2,1)= \max\{-1+0.5*0; -1+0.5*-1; -1+0.5*-1.5; -1+0.5*-1.5\}$
$V_4(2,2)= \max\{-1+0.5*-1; -1+0.5*-1; -1+0.5*-1.5; -1+0.5*-1.5\}$
$V_4(2,3)= \max\{-1+0.5*-1.5; -1+0.5*-1.5; -1+0.5*-1.5; -1+0.5*-1.5\}$

Stochastic Value Iteration

R=-1 for each action,, γ=0.5; 0.75 chance of changing state, 0..25 chance of staying in the same state

g			

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

$V_2(1,2)$= max{-1+0.5*(0.75*0 + 0.25*0);
-1+0.5*(0.75*0 + 0.25*0);
-1+0.5*(0.75*0 + 0.25*0);
-1+0.5*(0.75*0 + 0.25*0)}

Stochastic Value Iteration

R=-1 for each action,, γ=0.5; 0.75 chance of changing state, 0..25 chance of staying in the same state

g			

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	-1.125		

V_3

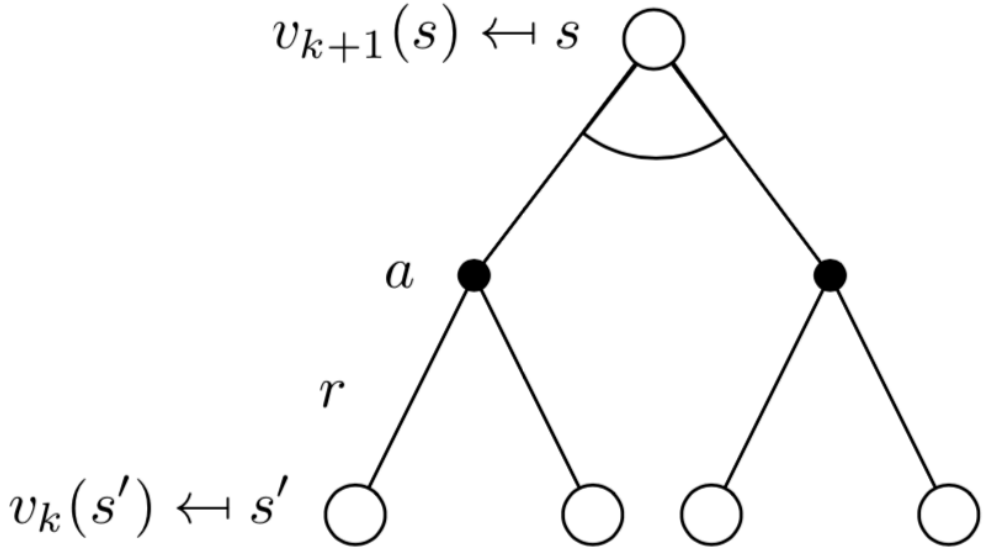
$V_3(1,2)$= max{-1+0.5*(0.75*0 + 0.25*-1); **Left Action:** 0.75 of going left, 0.25 os staying in 1,2
-1+0.5*(0.75*-1 + 0.25*-1); **North Action:** 0.75 of going north, 0.25 os staying in 1,2
-1+0.5*(0.75*-1 + 0.25*-1); **Right Action:** 0.75 of going right, 0.25 os staying in 1,2
-1+0.5*(0.75*-1 + 0.25*-1)} **South Action:** 0.75 of going south, 0.25 os staying in 1,2

$V_3(1,2)$=-1-0.125=-1.125

Iterative Policy Evaluation

- Problem: evaluate a given policy π
  Solution: iterative application of Bellman expectation backup
- $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_\pi$
- Using **synchronous** backups
  - At each iteration $k + 1$
  - For all states s ∈ S
  - Update $v_{k+1}(s)$ from $v_k(s')$
  - Where s' is a successor state of s
- It converges to v_π

Iterative Policy Evaluation



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

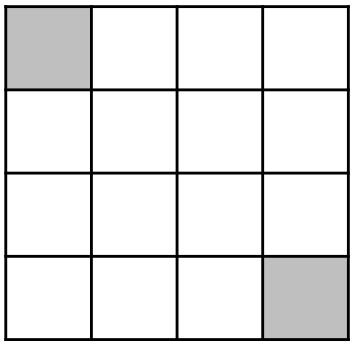
$$\mathbf{v}_{k+1}(s) = \boldsymbol{\mathcal{R}}^{\pi} + \gamma \boldsymbol{\mathcal{P}}^{\pi} \mathbf{v}_k$$

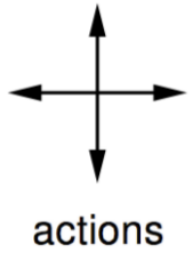
Evaluating a Random Policy in the Small Gridworld

- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states (1,2), ..., (4,3)
- Terminal states (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

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

- $r = -1$ on all transitions





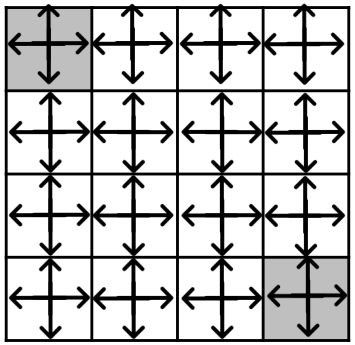
Iterative Policy Evaluation in Small Gridworld

V_k for random policy

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

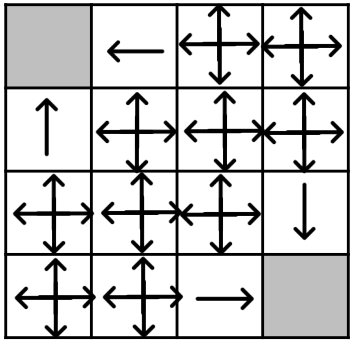
$k = 0$

Greedy policy w.r.t. v_k for random policy



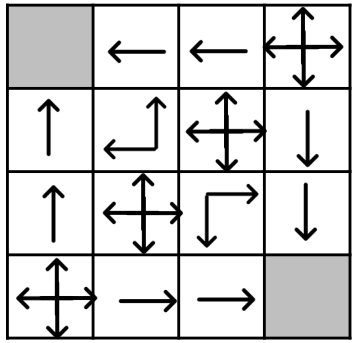
0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0

$k = 1$



0	-1.7	-2	-2
-1.7	-2	-2	--2
-2	-2	-2	-1.7
-2	-2	-1.7	0

$k = 2$



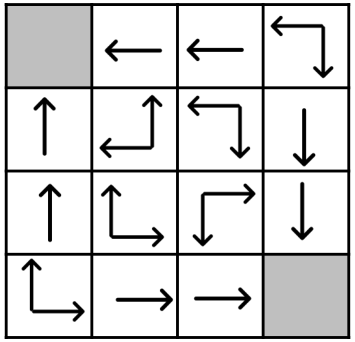
Iterative Policy Evaluation in Small Gridworld

V_k for random policy

0	-2.4	-2.9	-3
-2.4	-2.9	-3	-2.9
-2.9	-3	-2.9	-2.4
-3	-2.9	-2.4	0

$k = 3$

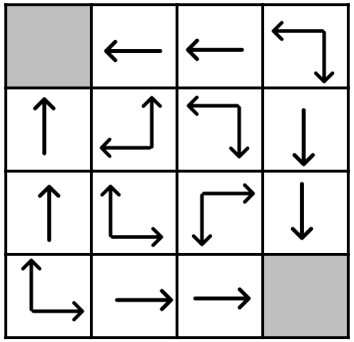
Greedy policy w.r.t. v_k for random policy



Optimal policy

0	-6.1	-8.4	-9
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9	-8.4	-6.1	0

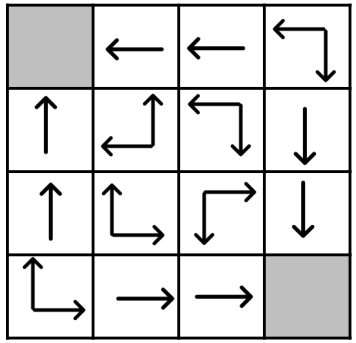
$k = 10$



Optimal policy

0	-14	-20	-22
-14	-18	-20	--20
-20	-20	-18	-1.4
-20	-20	-14	0

$k = \infty$



Optimal policy

Policy Iteration

- How to improve a policy?
- **Policy Iteration**
  - Given a policy π
    - Evaluate policy π

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]$$

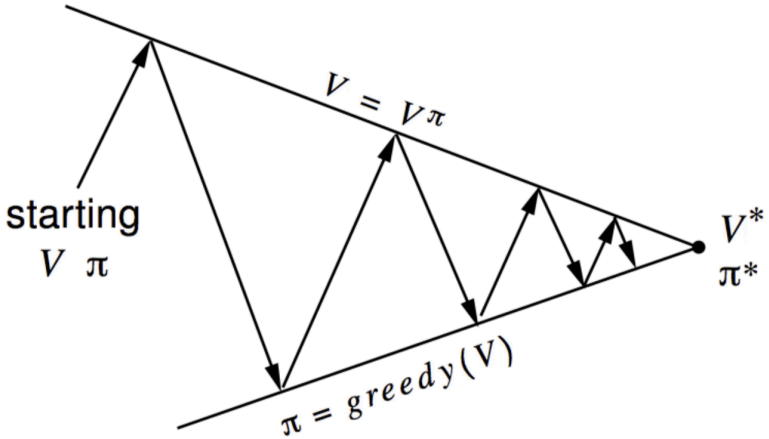
    - Improve policy π

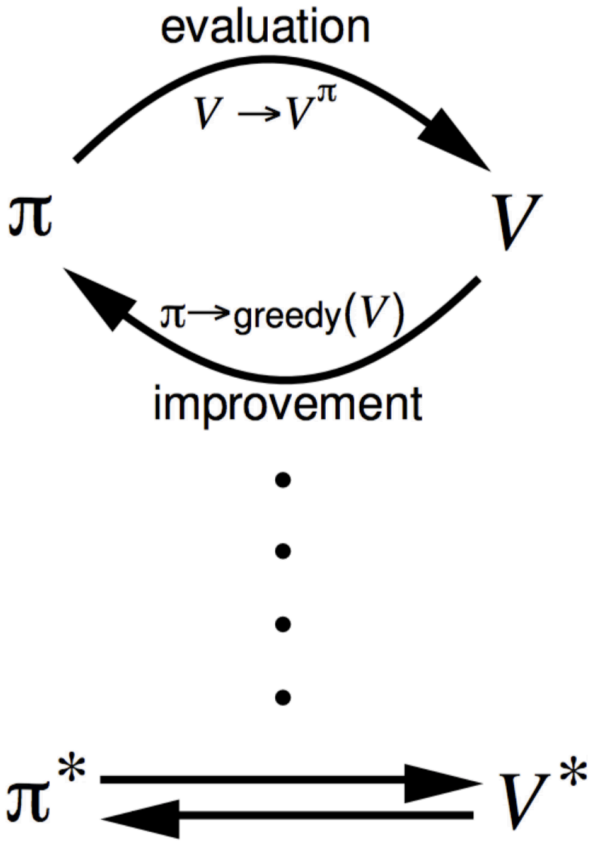
$$\pi' = greedy(v_\pi)$$

- 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 π_*

Policy Iteration

- **Policy Iteration**
  - Policy evaluation
    - Estimate v_π
      - Iterative policy evaluation
  - Policy improvement
    - Generate $\pi' \geq \pi$
      - Greedy policy improvement

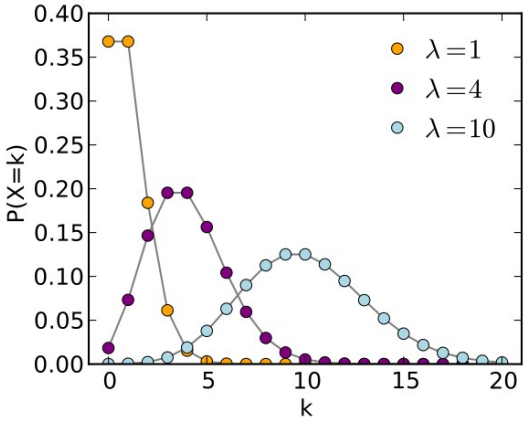




Policy Iteration

Jack's Car Rental

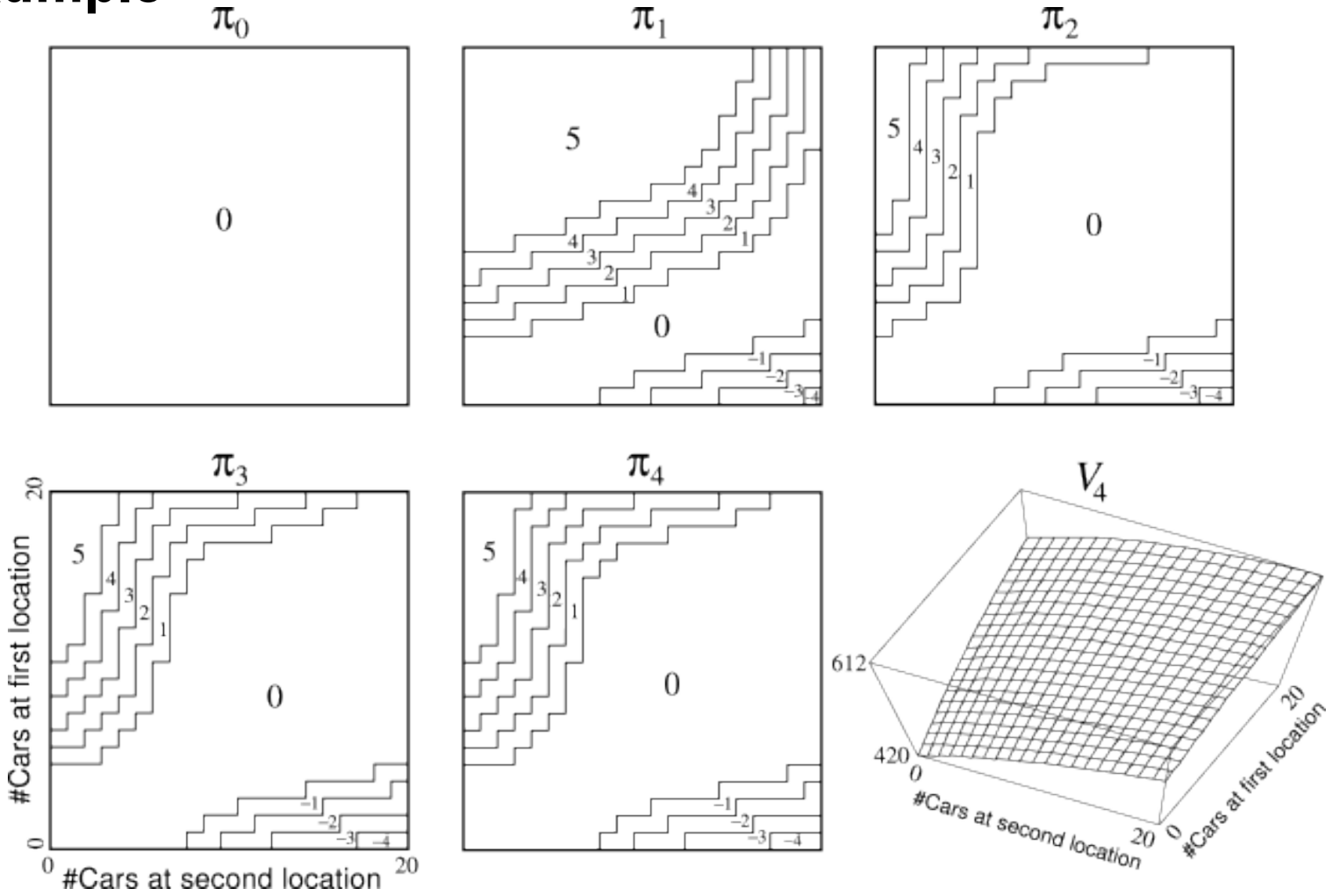
Example



- 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

- **Example**



Policy Improvement

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

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

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

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

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

$$v_\pi(s) \leq q_\pi\left(s, \pi'(s)\right) = \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | \, S_t = s]$$

$$\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi\left(S_{t+1}, \pi'(S_{t+1})\right) | \, S_t = s]$$

$$\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi\left(S_{t+2}, \pi'(S_{t+2})\right) | \, S_t = s]$$

$$\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \ldots | \, S_t = s] = v_{\pi'}(s)$$

Policy Improvement

- If improvements stop,
- We can improve the policy by acting greedily

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

- Then the Bellman optimality equation has been satisfied

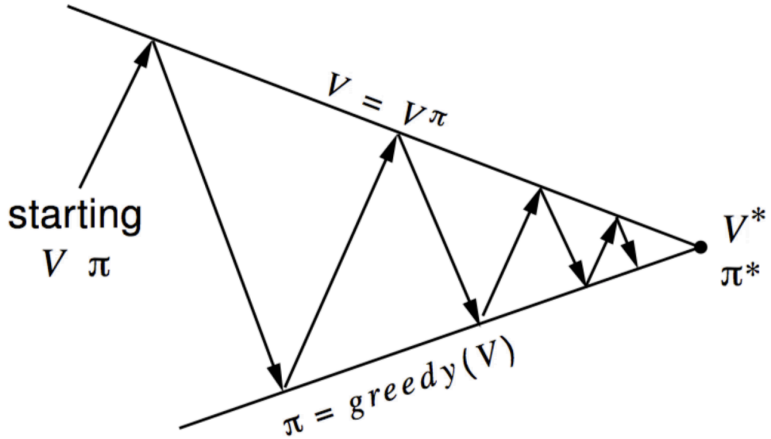
$$v_\pi(s) = \max_{a \in A} q_\pi(s,a)$$

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

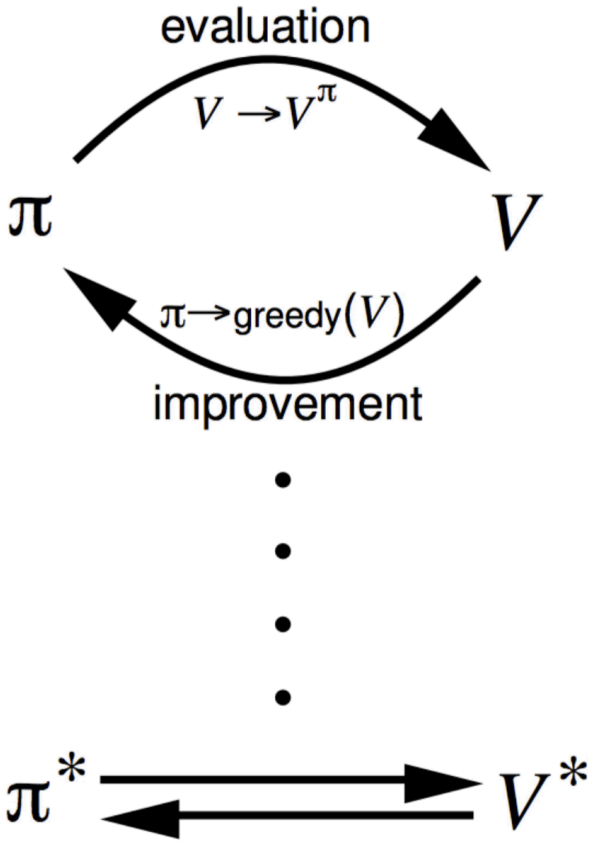
Policy Improvement

- Does policy evaluation need to converge to v_π?
- 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 $k=3$
  - This is equivalent to value iteration

Generalized Policy Iteration



- **Generalized Policy Iteration**
  - Policy evaluation
    - Estimate v_π
      - **Any** policy evaluation algorithm
  - Policy improvement
    - Generate $\pi' \geq \pi$
      - **Any** policy improvement



Synchronous Dynamic Programming Algorithms

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

- Algorithms are based on state-value function $v_\pi(s)$ or $v_*(s)$
  - Complexity $O(mn^2)$ per iteration, for m actions and n states
- Could also apply to action-value function $q_\pi(s|a)$ or $q_*(s|a)$
  - Complexity $O(m^2n^2)$ per iteration

Value iteration with Q

- The steps involved in the **value iteration** are as follows:
  1. First, we initialize the random value function, that is, the random value for each state.
  2. Then we compute the Q function for all state action pairs of $Q(s, a)$.
  3. Then we update our value function with the max value from $Q(s, a)$.
  4. We repeat these steps until the change in the value function is very small.

Asynchronous Dynamic Programming

- DP methods described so far used synchronous backups
  - i.e. all states are backed up in parallel
- Asynchronous DP backs up states individually, in any order
  - For each selected state, apply the appropriate backup
  - Can significantly reduce computation
  - Guaranteed to converge if all states continue to be selected

Asynchronous Dynamic Programming

- Three simple ideas for asynchronous dynamic programming:
  - In-place dynamic programming
  - Prioritized sweeping
  - Real-time dynamic programming

Asynchronous Dynamic Programming

- In-place dynamic programming
  - Synchronous value iteration stores two copies of value function
    - For all $s \in S$

$$v_{new}(s) = \max_{a \in A}\left(R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_{old}(s')\right)$$

$$v_{old} \leftarrow v_{new}$$

  - In-place value iteration only stores one copy of value function
    - For all $s \in S$

$$v(s) = \max_{a \in A}\left(R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v(s')\right)$$

Asynchronous Dynamic Programming

- Prioritized Sweeping
  - Use magnitude of Bellman error to guide state selection, e.g.

$$\left| \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v(s') \right) - v(s) \right|$$

- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue

Asynchronous Dynamic Programming

- Real-Time Dynamic Programming
  - Idea: only states that are relevant to agent
  - Use agent's experience to guide the selection of states
  - After each time-step S_t, A_t, R_{t+1}
  - Backup the state S_t

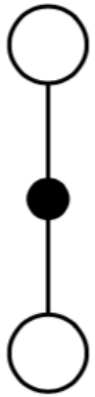
$$v(S_t) = \max_{a \in A} \left(R^a_{S_t} + \gamma \sum_{s' \in S} \mathcal{P}^a_{S_t s'} v(s') \right)$$

Full-Width Backups

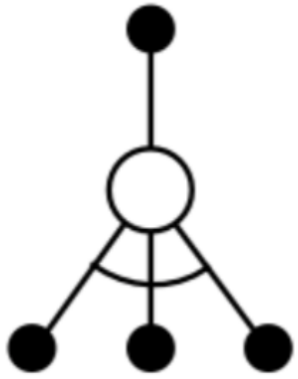
- DP uses full-width backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's curse of dimensionality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

Sample Backups

- What if we don't have $\mathcal{R}$ and $\mathcal{P}$
  - We will employ model-free algorithms
  - Consider sample backups
  - Using sample rewards and sample transitions MDP $\langle S, A, \mathcal{R}, S' \rangle$ instead of reward function $\mathcal{R}$ and transiti 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 = $|S|$



- 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 $\tilde{S} \subseteq S$
  - For each state $s \in \tilde{S}$, estimate target value using Bellman optimality equation,

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

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

Relevant points

- How do we know that value iteration converges to v_*?
- Or that iterative policy evaluation converges to v_π?
- And therefore that policy iteration converges to v_*?
- Is the solution unique?
- How fast do these algorithms converge?
- These questions are resolved by contraction mapping theorem

Value Function Space

- Consider the vector space V over value functions
- There are $|S|$ dimensions
- Each point in this space fully specifies a value function $v(s)$
- What does a Bellman backup do to points in this space?
- We will show that it brings value functions closer
- And therefore the backups must converge on a unique solution

Value Function ∞-Norm

- We will measure distance between state-value functions u and v by the ∞-norm
- i.e. the largest difference between state values,

$$||u - v||_\infty = \max_{s \in S} |u(s) - v(s)|$$

Bellman Expectation Backup is a Contraction

- Define the Bellman expectation backup operator T^{π},

$$T^{\pi}(v) = R^{\pi} + \gamma P^{\pi} v$$

- This operator is a γ-contraction, i.e. it makes value functions closer by at least γ,

$$\begin{aligned} ||T^{\pi}(u) - T^{\pi}(v)||_{\infty} &= \left|\left|(R^{\pi}+\gamma P^{\pi}u) - (R^{\pi}+\gamma P^{\pi}v)\right|\right|_{\infty} \\ &= ||\gamma P^{\pi}(u-v)||_{\infty} \\ &\leq \left|\left|\gamma P^{\pi}||u-v||_{\infty}\right|\right|_{\infty} \\ &\leq \gamma||u-v||_{\infty} \end{aligned}$$

Contraction Mapping Theorem

- Theorem (Contraction Mapping Theorem)

- For any metric space V that is complete (i.e. closed) under an operator $T \quad (v)$, where T is a γ-contraction,
  - T converges to a unique fixed point
  - At a linear convergence rate of γ

Convergence of Iter. Policy Evaluation and Policy Iteration

- The Bellman expectation operator T^π has a unique fixed point
- v_π is a fixed point of T^π (by Bellman expectation equation)
- By contraction mapping theorem
  - Iterative policy evaluation converges on v_π
  - Policy iteration converges on v_*

Bellman Optimality Backup is a Contraction

- Define the Bellman optimality backup operator T^*,

$$T^*(v) = \max_{a \in A} R^a + \gamma P^a v$$

- This operator is a γ-contraction, i.e. it makes value functions closer by at least γ (similar to previous proof)

$$||T^*(u) - T^*(v)||_\infty \leq \gamma ||u - v||_\infty$$

Convergence of Value Iteration

- The Bellman optimality operator T^* has a unique fixed point
- v_* is a fixed point of T^* (by Bellman optimality equation)
- By contraction mapping theorem
- Value iteration converges on v_*

Implementations

- Check the notebooks available to test the Policy Iteration and Value Iteration Algorithms
  - Compute Policy Iteration for the **q function**
  - Make the environment stochastic in value iteration. Does it change?
  - Change γ in both methods and evaluate the implications
  - Implement variations of the methods to discuss how you could improve the results in terms of computation time

Lecture 3

- Reading:
  - RUSSELL, S. NORVIG, P. Artificial Intelligence. 3a edição. Chapter 21.
  - BARTO, A., SUTTON, R. Reinforcement Learning: An Introduction. Second Edition. Freely Available at: https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view?usp=sharing

References

Lecture 2

- BARTO, A., SUTTON, R. Reinforcement Learning: An Introduction. Second Edition.
- MURPHY, R. R. Introduction to AI robotics. MIT Press, 2002.
- Lex Fridman, MIT Deep Learning Course, MIT, 2019.
- DUDEK, G.; JENKIN, M. Computational Principles of mobile robotics. Cambridge Press, 2000.
- ROMERO, R. A. F.; PRESTES, E.; OSÓRIO, F.; WOLF, D. (Orgs) Robótica móvel. LTC, 2014.
- BROOKS, R. Intelligence without representation. Artificial Intelligence, 47:139-159, 1991.
- RUSSEL, S. NORVIG, P. Artificial Intelligence: a modern approach. Prentice Hall, 2002.
- BRATKO, I. PROLOG: programming for artificial intelligence. Addison Wesley, 2nd edition, 1990.

This material is part of the Machine Learning Course

By Esther Colombini and Alexandre Simões