Lecture 18: Markov Decision Processes (II)

EMAT31530/April 2018/Raul Santos-Rodriguez

Have a look at ...

... Russell and Norvig (Ch. 17 and Ch. 21)

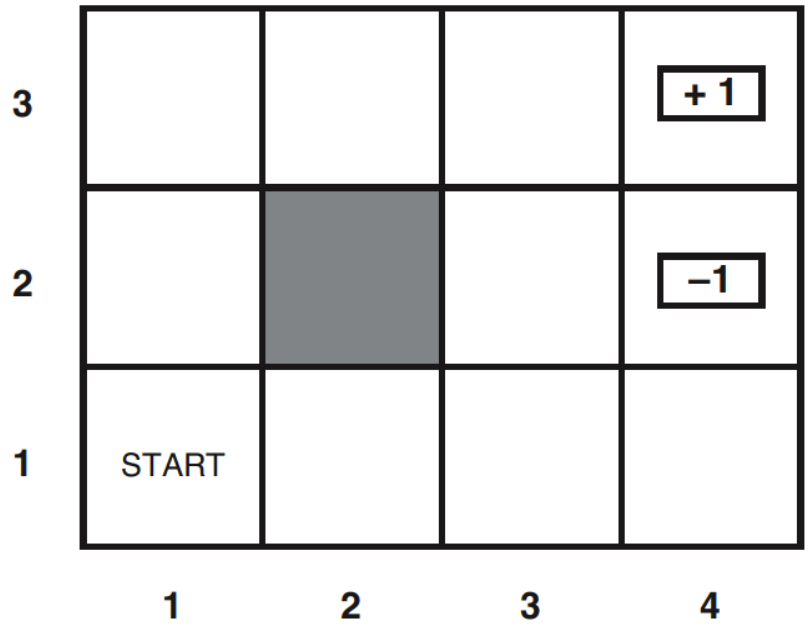
Outline

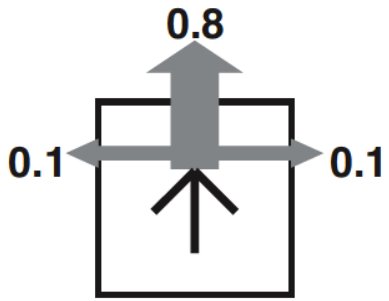
This lecture continues with the discussion on complex decision making. The objective is to present two alternatives to deal with Markov Decision Processes:

- Value iteration
- Policy iteration

Finally, we will introduce the concept of Reinforcement Learning.

Example





MDP formulation

MDP requires a structure to keep track of the decision sequences:

MDP

- s: state
- Actions(s): possible actions
- $P(s'|s,a)$ (or $T(s,a,s')$): probability of s' if take action a in state s
- Reward(s): reward for the state s
- Goal(s): whether at the end of the process
- $U_h([s_1, s_2, \ldots])$: utility or value of a sequence of states
- $\pi(s)$: policy describing which action should be taken in s

Optimal policies

Expected utility/value

The expected utility obtained by executing π starting in s is given by

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R(s_t)\right]$$

Optimal policies

Out of all the policies we could choose to execute starting in s, one will have higher expected utilities than all the others. This is an optimal policy when s is the starting state,

$$\pi_s^* = \arg\max_{\pi} U^{\pi}(s)$$

Optimal policies

Expected utility/value

The expected utility obtained by executing π starting in s is given by

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

Optimal policies

Out of all the policies we could choose to execute starting in s, one will have higher expected utilities than all the others. This is an optimal policy when s is the starting state,

$$\pi_s^* = \arg\max_{\pi} U^{\pi}(s)$$

Having decided that the utility of a given state sequence is the sum of discounted rewards obtained during the sequence, we can compare policies by comparing the expected utilities obtained when executing them. We assume the robot is in some initial state s and define s_t (a random variable) to be the state the robot reaches at time t when executing a particular policy π. (Obviously, $s_0 = s$, the state the robot is in now.) The probability distribution over state sequences $s_1, s_2, \ldots$, is determined by the initial state s, the policy π, and the transition model for the environment.

Be careful with the notation: remember that π_s^* is a policy, so it recommends an action for every state; its connection with s in particular is that it's an optimal policy when s is the starting state.

Optimal policies

Discounted utilities + infinite horizons $\rightarrow$ optimal policy is independent of the starting state

Note

If policy π_a^* is optimal starting in state a and policy π_b^* is optimal starting in b, then, when they reach a third state c, there's no good reason for them to disagree with each other, or with π_c^*, about what to do next. So we will write π^* for an optimal policy.

True utility of a state is $U^{\pi^*}(s)$ or just $U(s)$: *expected* sum of *discounted rewards* if the robot executes an *optimal policy*.

Optimal policies

Optimal policies

Discounted utilities + infinite horizons → optimal policy is independent of the starting state

Note

If policy π_a^* is optimal starting in state a and policy π_b^* is optimal starting in b, then, when they reach a third state c, there's no good reason for them to disagree with each other, or with π_c^*, about what to do next. So we will write π^* for an optimal policy.

True utility of a state is $U^{\pi^*}(s)$ or just $U(s)$: *expected* sum of *discounted rewards* if the robot executes an *optimal policy*.

Although using discounted utilities with infinite horizons implies that the optimal policy is independent of the starting state, the action sequence won't be independent; remember that a policy is a function specifying an action for each state.

Value iteration

Idea: Calculate the utility of each state and then select optimal action based on these utilities

Criterion for optimal policy

Assuming discounted rewards,

$$\pi^* = \arg\max_{\pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi\right]$$

Value iteration

Value iteration

Idea: Calculate the utility of each state and then select optimal action based on these utilities

Criterion for optimal policy

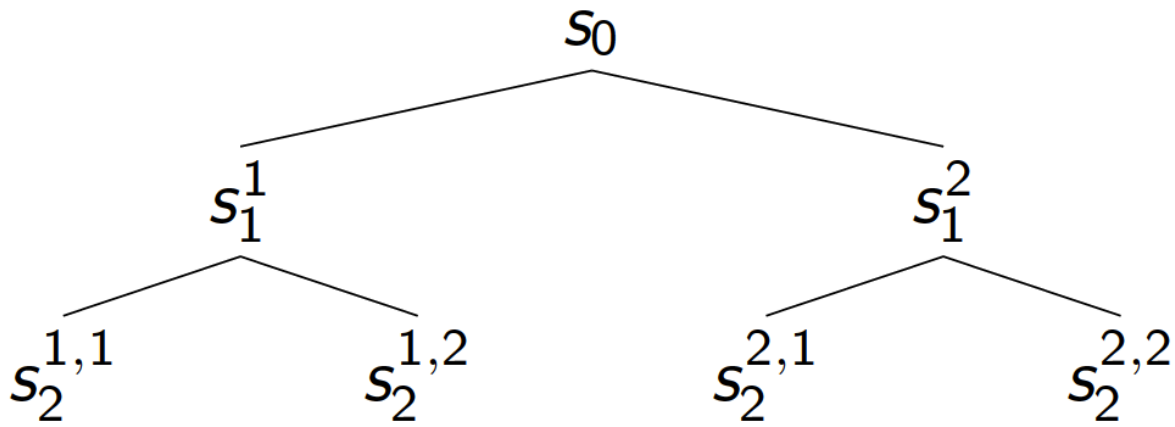
Assuming discounted rewards,

$$\pi^* = \arg\max_\pi E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi\right]$$

Value iteration is an algorithm for calculating the optimal policy in MDPs.

Value iteration: criterion

Each policy π yields a tree, with root node s_0, where the children of node s are the possible successor states given the action $\pi(s)$.



$P(s'|a, s)$ gives the probability of traversing an edge from s to s'.

The **expectation** involves two steps:

1. For each path in the tree, getting the product of the (joint) probability of the path in this tree with its discounted reward, and then
2. Summing over all the products from (1)

Value iteration: utilities of states

Some remarks

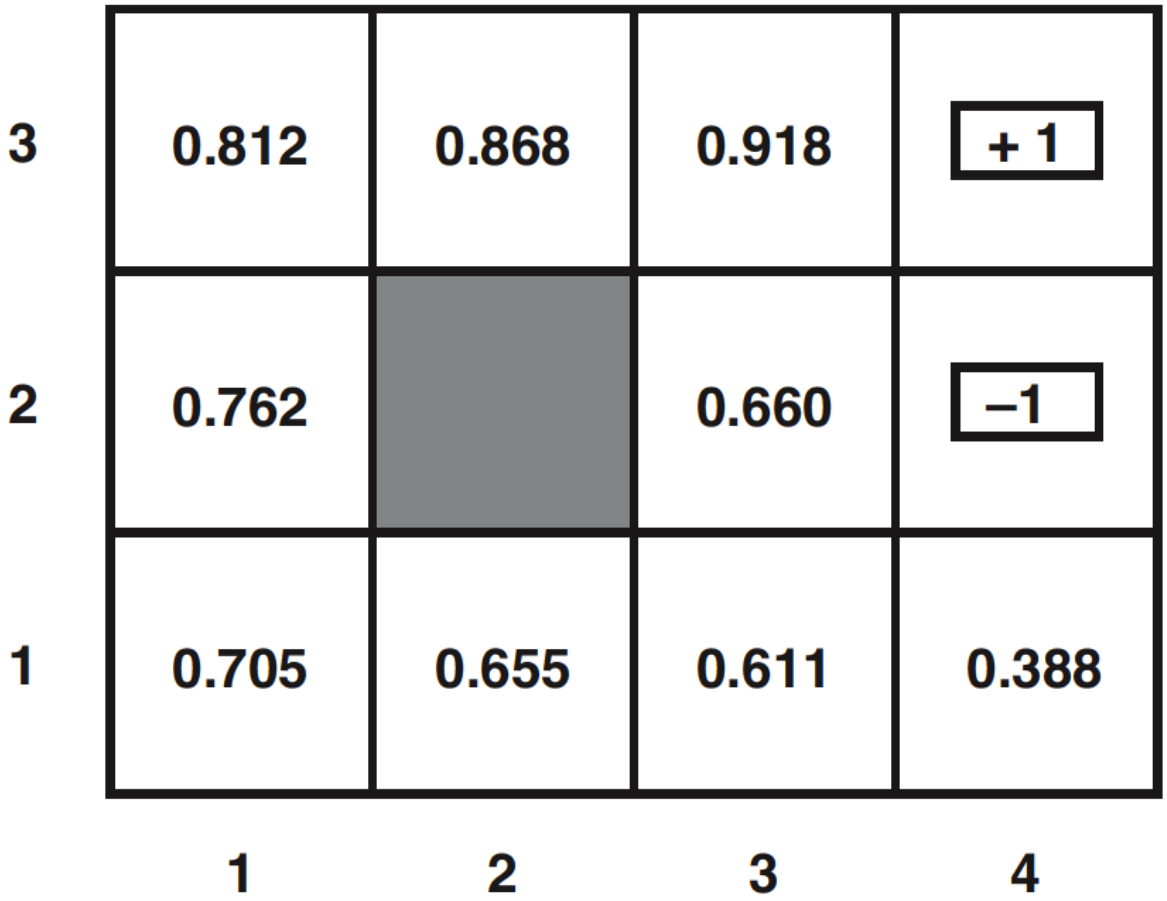
- $R(s)$ is reward for being in s *now*: is the short term reward for being in s
- $U(s)$ is utility of the states that might follow s: $U(s)$ captures long term advantages from being in s
- $U(s)$ reflects what you can do from s;
- $R(s)$ does not.

States that follow depend on π. Utility of s given π is,

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right]$$

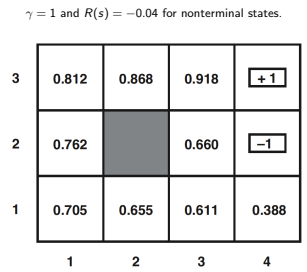
Value iteration: example

$\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.



Value iteration: example

Value iteration: example



The figure shows the utilities for the $4x3$ world. Notice that the utilities are higher for states closer to the $+1$ exit, because fewer steps are required to reach the exit.

Value iteration: Bellman equations

Optimal policy

Given $U(s)$, we can easily determine the optimal policy

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|a,s)U(s') \tag{1}$$

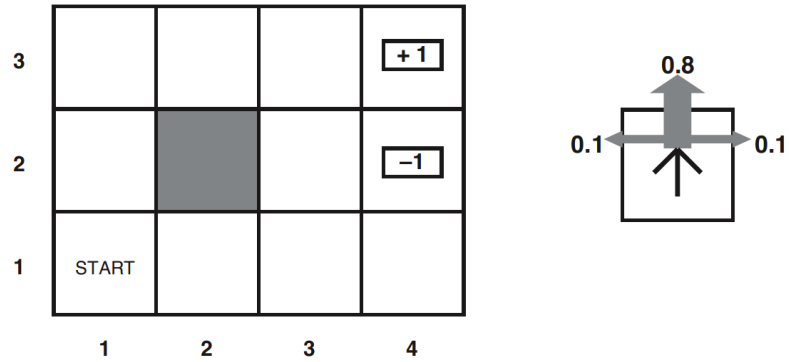
There is a direct relationship between the utility of a state and the utility of its neighbours.

Bellman equations (1957)

Utility of a state is the immediate reward plus expected utility of subsequent states if we choose the optimal action,

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|a,s)U(s')$$

Value iteration: example



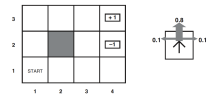
Bellman equation for the state (1,1) is

$$U(1,1) = -0.04 + \gamma \max\{0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), (Up)$$
$$0.9U(1,1) + 0.1U(1,2), (Left)$$
$$0.9U(1,1) + 0.1U(2,1), (Down)$$
$$0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1), (Right)\}$$

***Up* is the best action**

Value iteration: example

Value iteration: example



Bellman equation for the state (1,1) is

$$U(1,1) = -0.04 + \gamma \max\{0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), (Up)$$
$$0.9U(1,1) + 0.1U(1,2), (Left)$$
$$0.9U(1,1) + 0.1U(2,1), (Down)$$
$$0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1), (Right)\}$$

***Up* is the best action**

Using the values from the figure in slide 11, we can see that *Up* is the best action.

Value iteration: algorithm

For n states we have n Bellman equations with n unknowns (utilities of states)

Value iteration is an *iterative* approach to solving the n equations.

Intuition

We start with arbitrary values and update them as follows

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|a,s) U_i(s')$$

The algorithm converges to right and unique solution.

Value iteration: algorithm

Value iteration: algorithm

For n states we have n Bellman equations with n unknowns (utilities of states)

Value iteration is an *iterative* approach to solving the n equations.

Intuition

We start with arbitrary values and update them as follows

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|a,s) U_i(s')$$

The algorithm converges to right and unique solution.

The value iteration algorithm can be though as propagating values through the network of utilities.

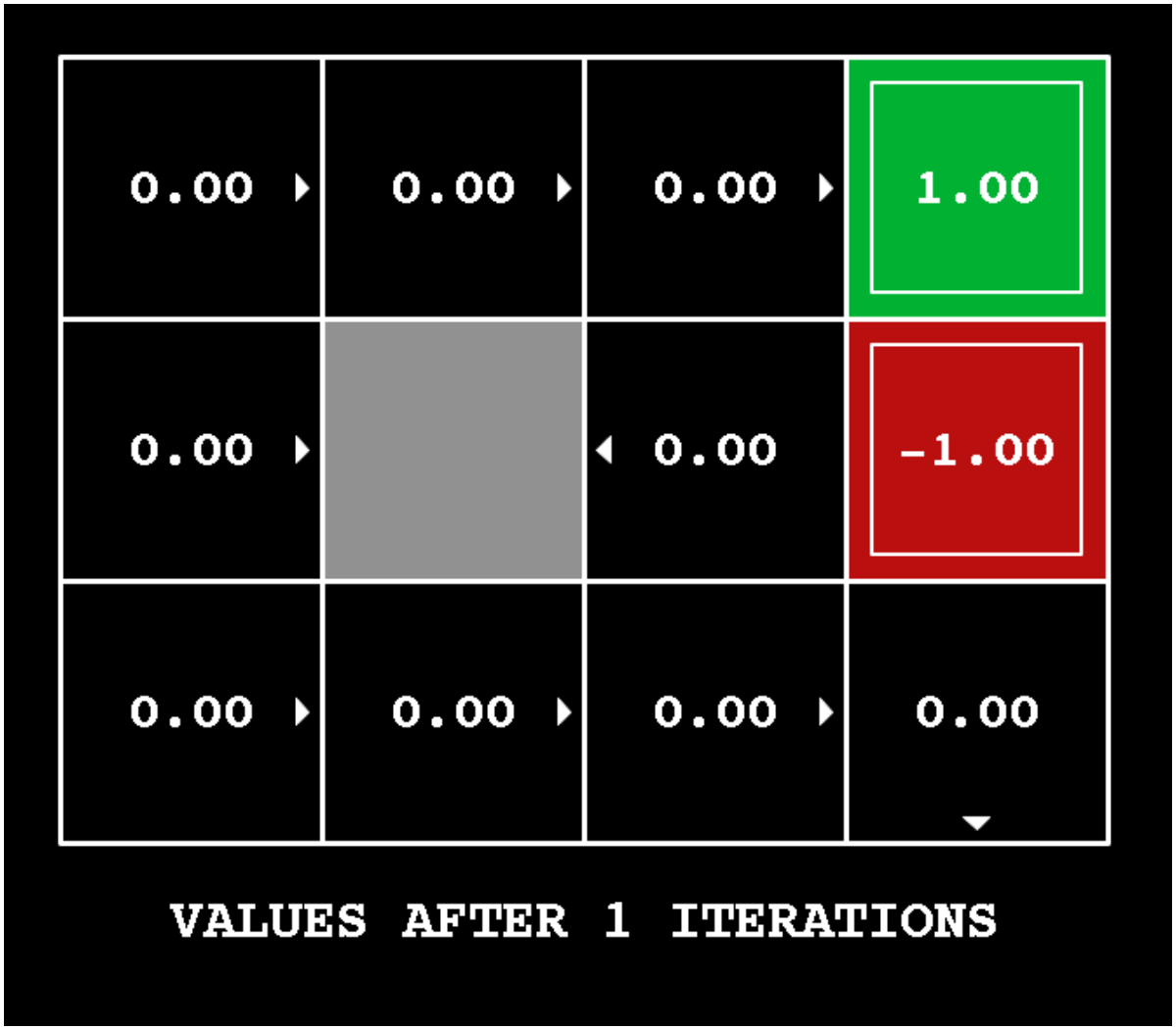
Value iteration: algorithm

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

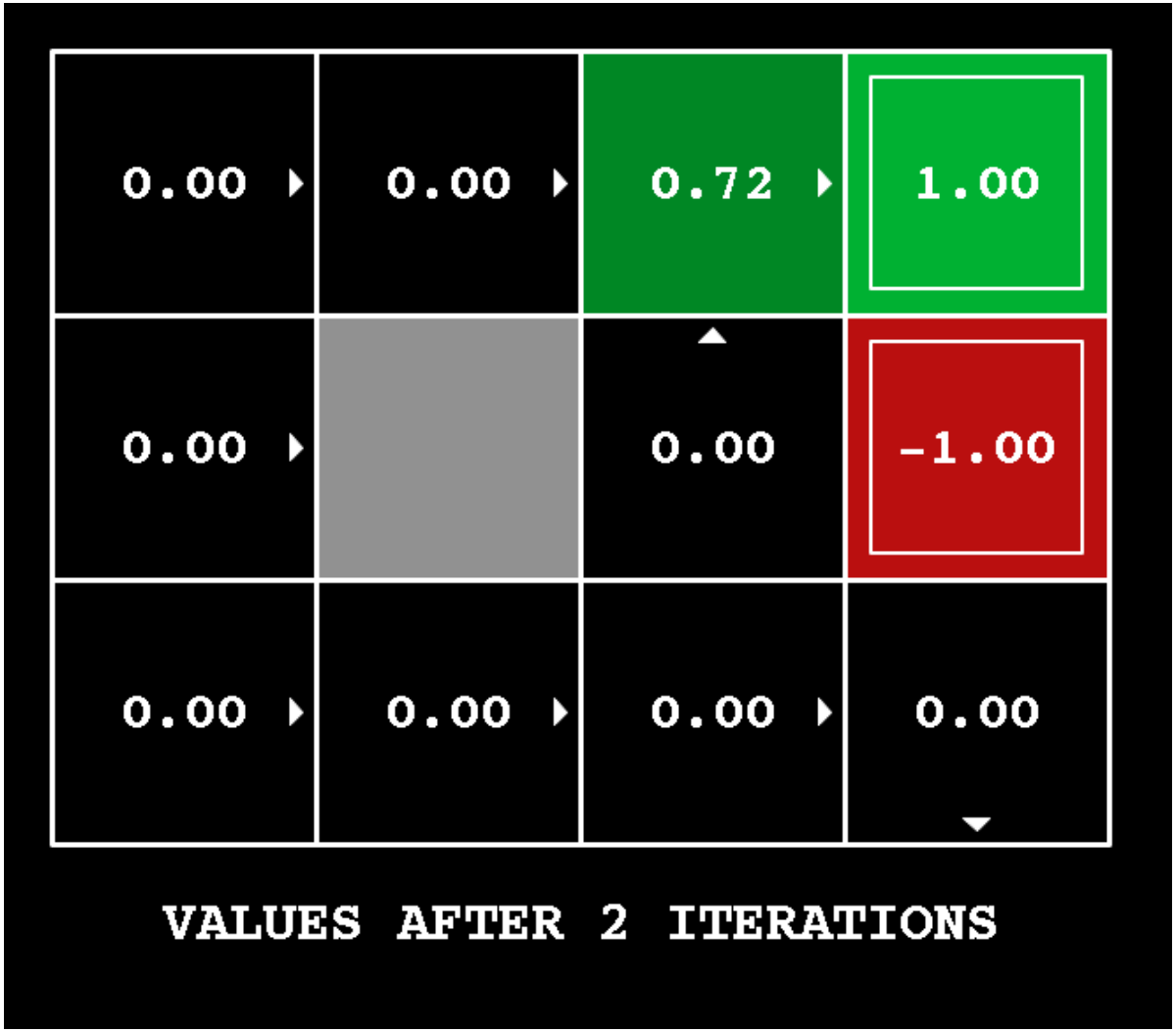
Value iteration: example

4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states



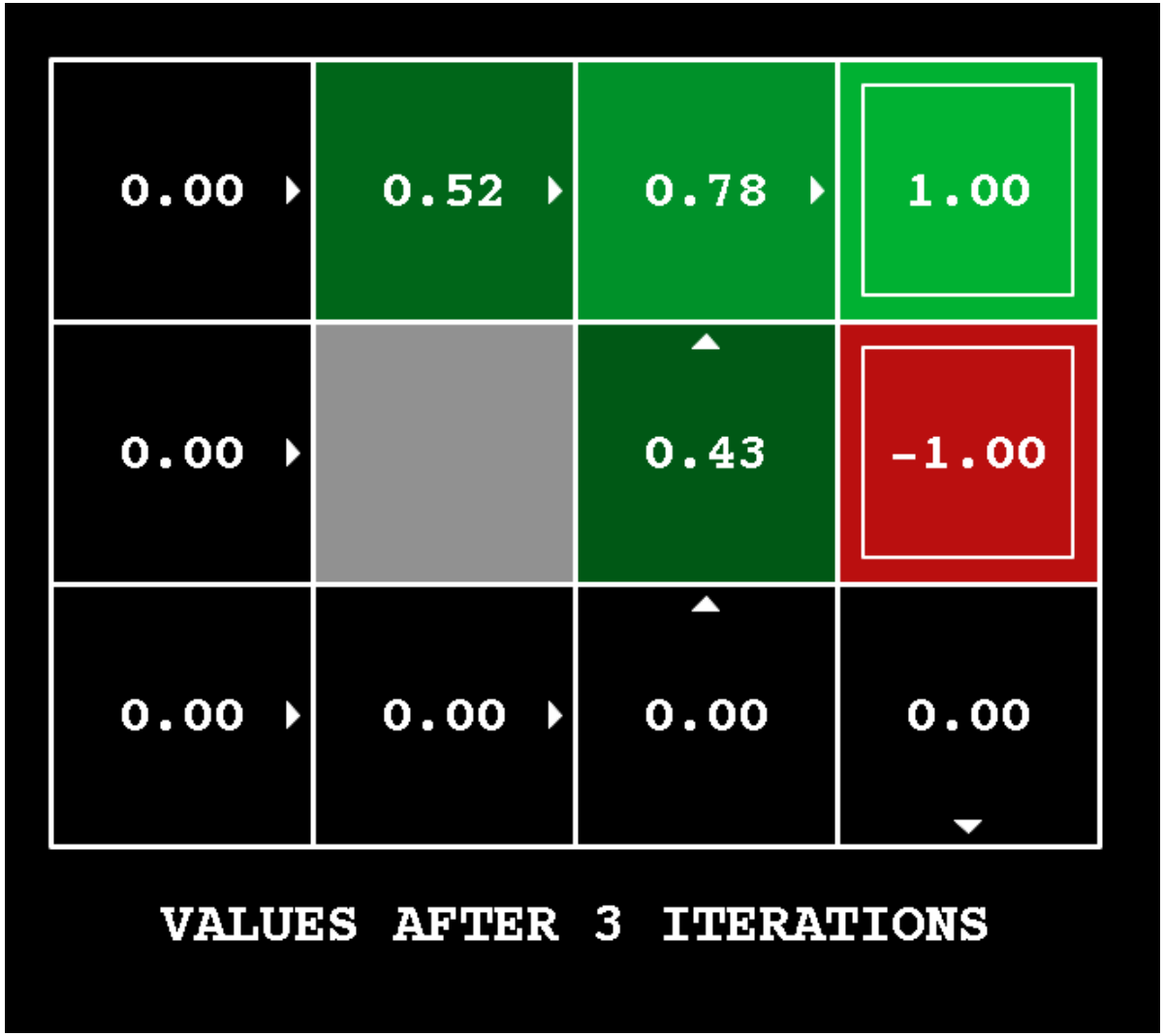
Value iteration: example

4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states



Value iteration: example

4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states

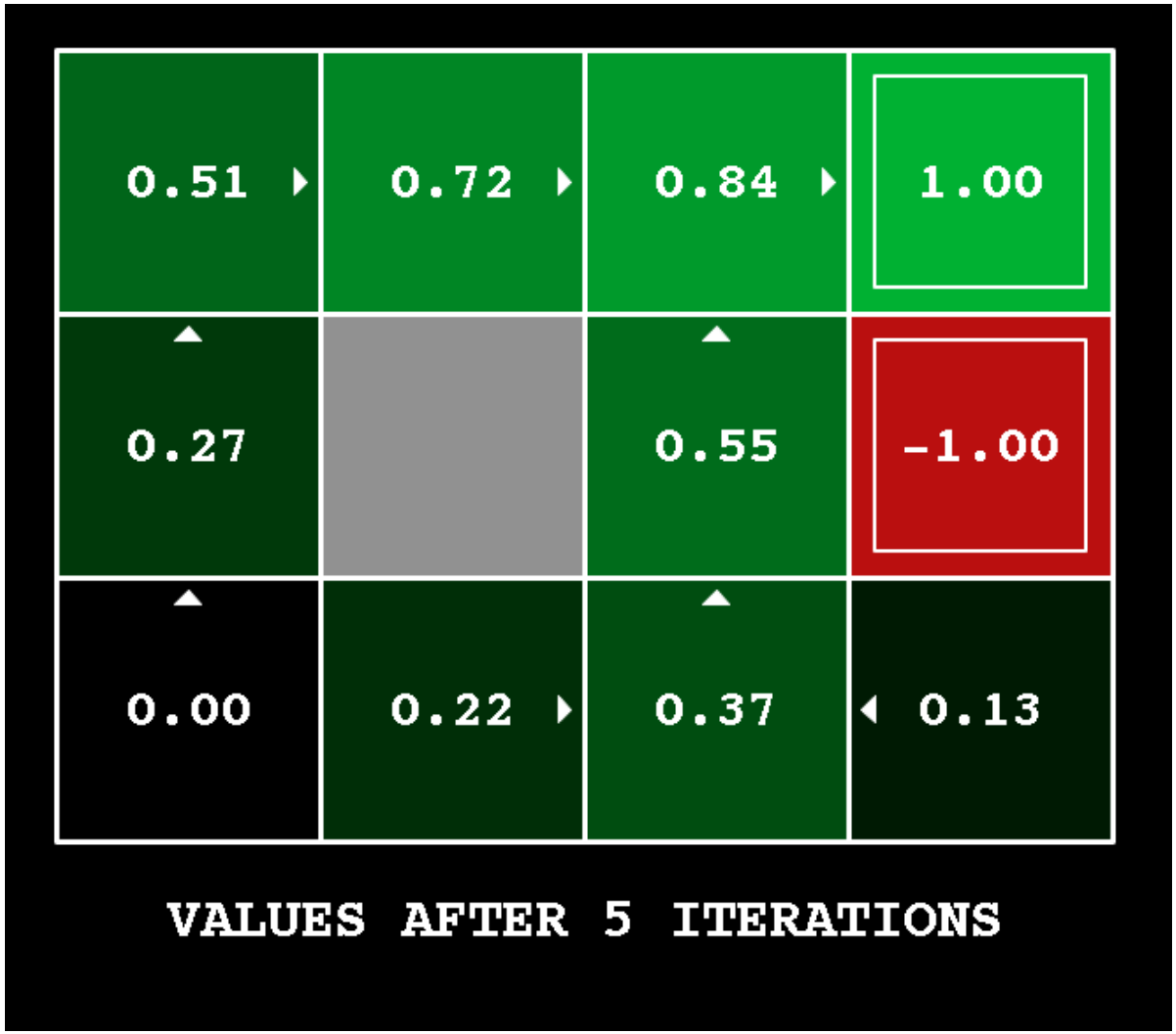


Value iteration: example

4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states

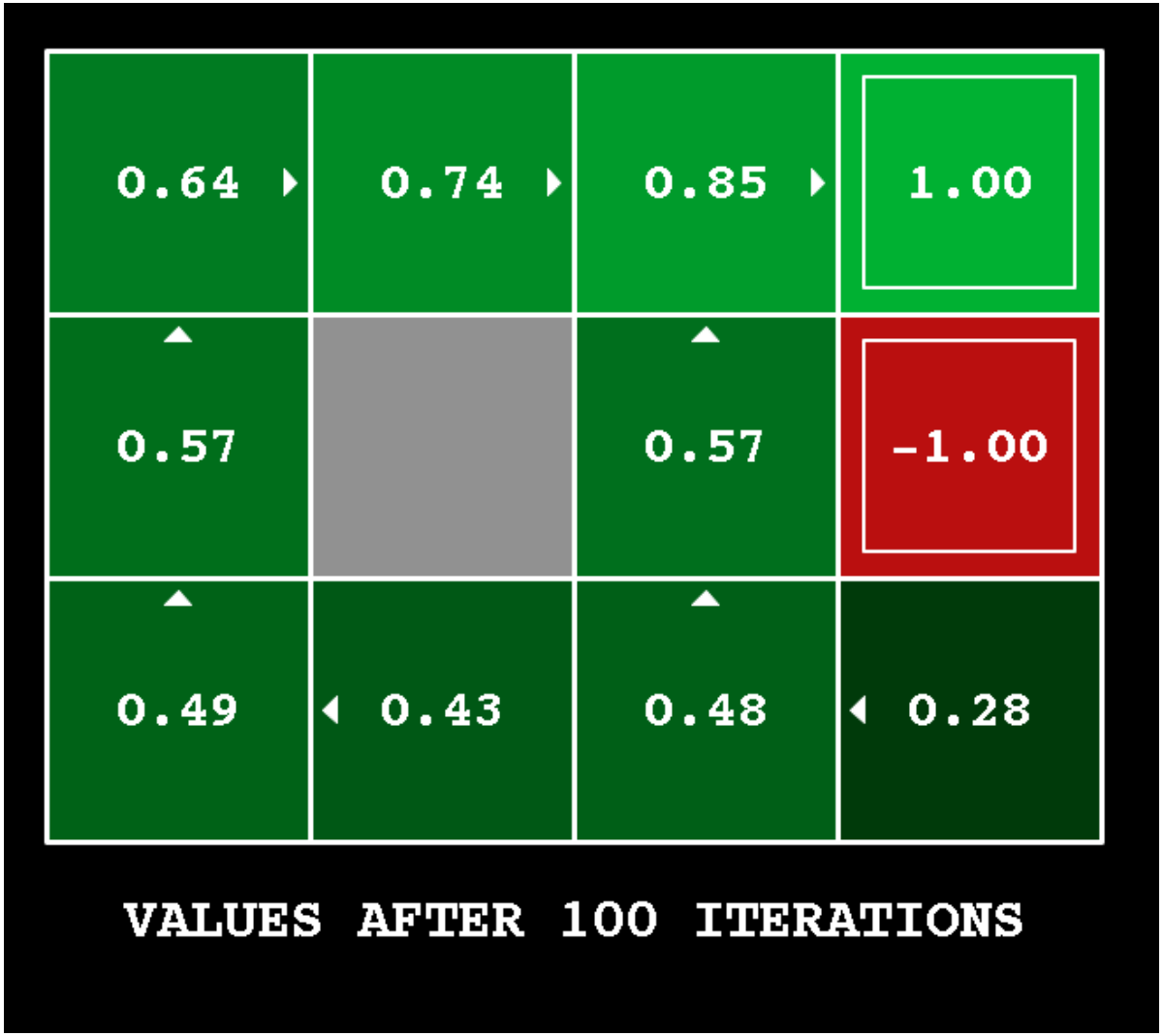
Value iteration: example

4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states



Value iteration: example

4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states

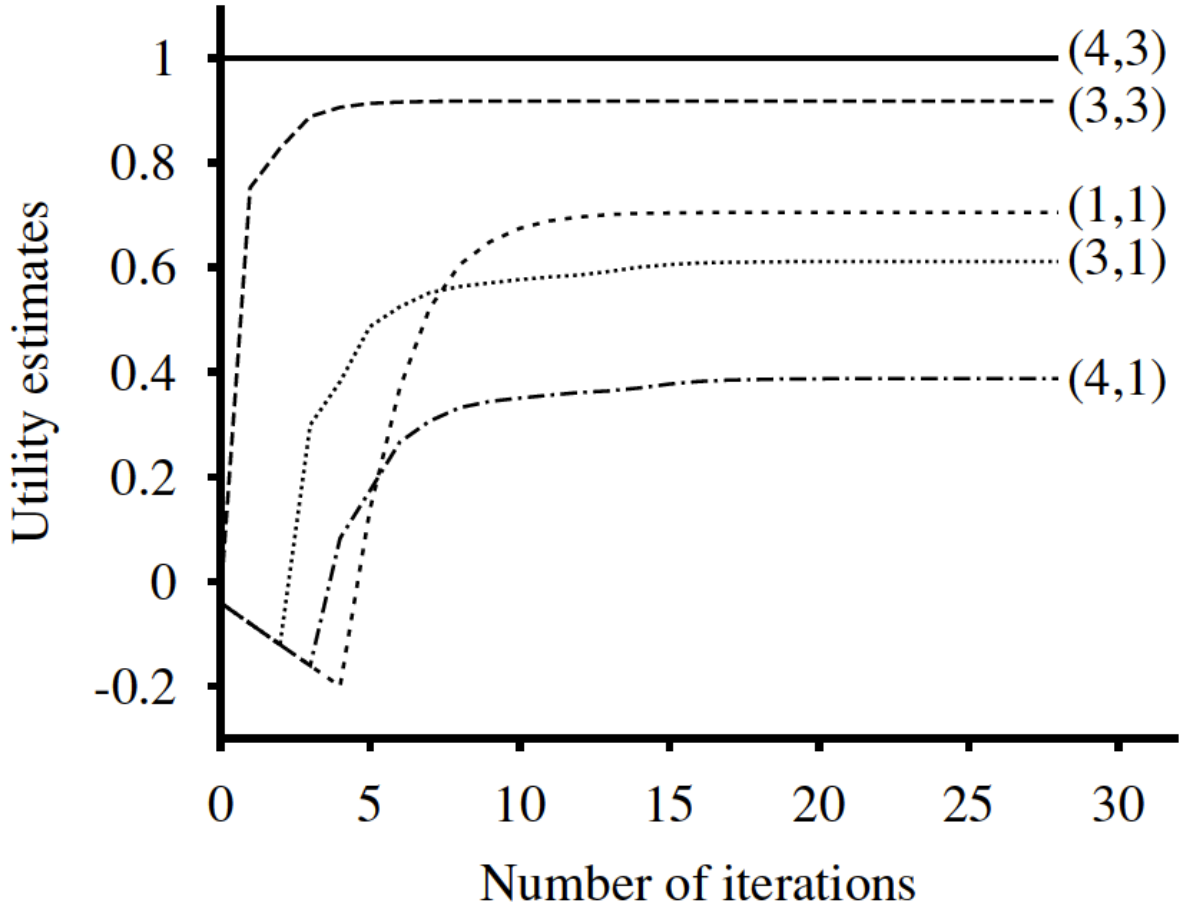


Value iteration: example

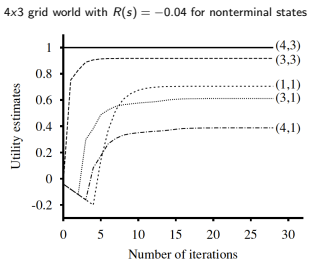
4x3 grid world with $\gamma = 0.9$ and $R(s) = 0$ for nonterminal states

Value iteration: algorithm

4x3 grid world with $R(s) = -0.04$ for nonterminal states



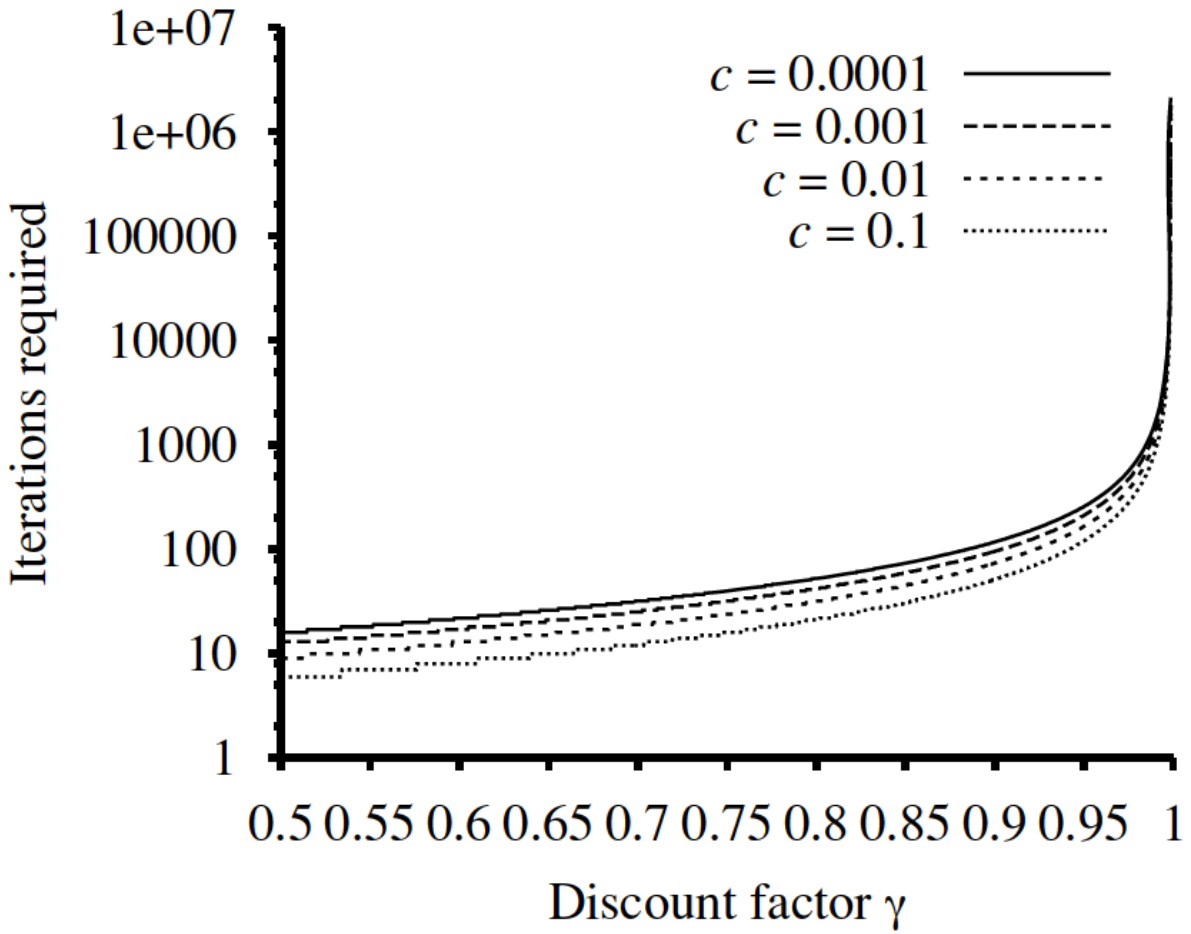
Value iteration: algorithm



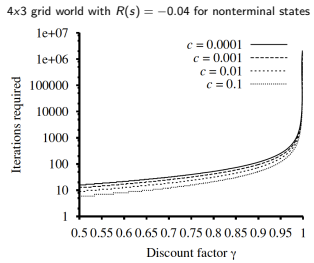
Graph showing the evolution of the utilities of selected states using value iteration.

Value iteration: algorithm

4x3 grid world with $R(s) = -0.04$ for nonterminal states



Value iteration: algorithm



The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{max}$, for different values of c, as a function of the discount factor γ.

Policy iteration

Idea: if one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise.

Algorithm

Policy iteration alternates two steps, beginning from some initial policy π_0:

1. **Policy evaluation**: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.
2. **Policy improvement**: Calculate a new MEU policy π_{i+1}, using one-step look-ahead based on U_i (using Eq. 1).

The algorithm terminates when the policy improvement step yields no change in the utilities.

Policy iteration

Idea: if one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise.

Algorithm

Policy iteration alternates two steps, beginning from some initial policy π_0:

- **Policy evaluation**: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.
- **Policy improvement**: Calculate a new MEU policy π_{i+1}, using one-step look-ahead based on U_i (using Eq. 1).

The algorithm terminates when the policy improvement step yields no change in the utilities.

Note: when the algorithm terminates, we know that the utility function U_i is a fixed point of the Bellman update, so it is a solution to the Bellman equations, and π_i must be an optimal policy. Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate.

Policy iteration: policy evaluation

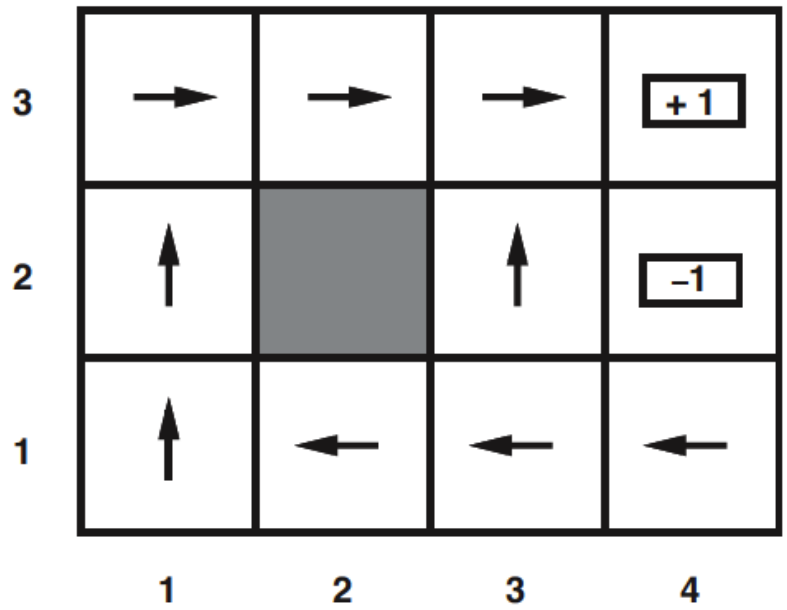
Policy improvement is easy, but policy evaluation?

Policy evaluation

- Simpler than solving the standard Bellman equations: the action in each state is fixed by the policy.
- At the ith iteration, the policy π_i specifies the action $\pi_i(s)$ in state s: simplified version of the Bellman equation relating the utility of s (under π_i) to the utilities of its neighbours:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

Policy iteration: example



E.g. $\pi_i(1,1) = Up$, $\pi_i(1,2) = Up$, ...

The simplified Bellman equations are

$$U_i(1,1) = -0.04 + 0.8U_i(1,2) + 0.1U_i(1,1) + 0.1U_i(2,1),$$

$$U_i(1,2) = -0.04 + 0.8U_i(1,3) + 0.2U_i(1,2),$$

...

Policy iteration: algorithm

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ **true**
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
        $unchanged? \leftarrow$ **false**
  until $unchanged?$
  return π

Policy iteration vs Value iteration

The equations are now **linear**: the max operator has been removed.

For n states, we have n linear equations with n unknowns, which can be solved exactly in time $O(n^3)$ by standard linear algebra methods.

When to use Policy iteration?

- For small state spaces: **policy evaluation** using exact solution methods is often the most efficient approach, typically very fast and converges quickly.
- For large state spaces, $O(n^3)$ time might be prohibitive. **Value iteration** is preferred.
- For very large state spaces: use an **approximation** but optimality guarantee is lost.

Summary

The road so far

- We now have (tediously) gathered all the ingredients to build MDPs.

- Transition models are described by a DBN

- Decisions will be made by projecting forward possible action sequences and choosing the best one.

MDPs are great, if

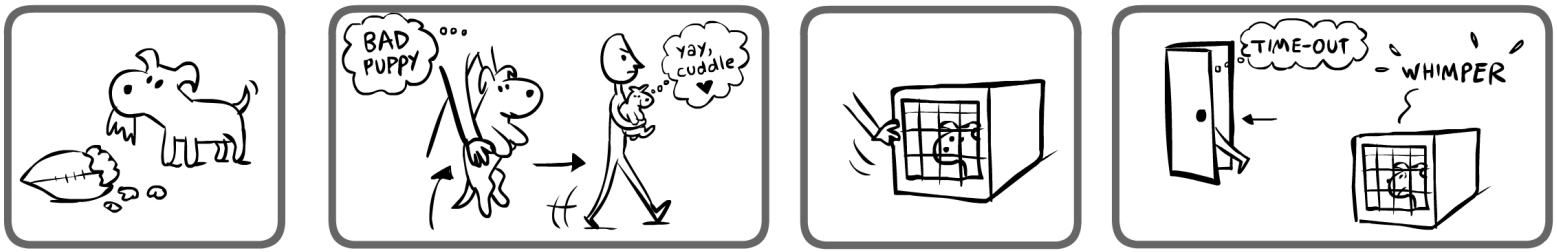
... we know the state transition function $P(s'|a, s)$

... we know the reward function $R(s)$

But what if we don't?

Like when we were babies

Reinforcement learning: demo



What we think the puppy is learning: **CHEWING UP STUFF IS BAD.**
What the puppy is really learning: **CHEWING UP STUFF is AWESOME. CRATE is BAD.**

DEMO

[https://www.youtube.com/user/stanfordhelicopter]

Reinforcement learning

- Actions have non-deterministic effects: initially unknown
- Rewards / punishments are infrequent and often at the end of long sequences of actions
- Learner must decide what actions to take
- World is large and complex

Reinforcement learning

Reinforcement learning is **on-line** methodology that we use when the **model of world is unknown** and/or **rewards are delayed**.

Reinforcement learning: Naive approach

Naive Approach

1. Act randomly for a while (or systematically explore all possible actions)
2. Learn Transition model and Reward function
3. Use value iteration, policy iteration, ...

Problems?

Reinforcement learning: Naive approach

Reinforcement learning: Naive approach

Naive Approach

- Act randomly for a while (or systematically explore all possible actions)
- Learn Transition model and Reward function
- Use value iteration, policy iteration, ...

Problems?

The naive approach involves selecting an action and then:
- If action leads to a reward, reinforce that action.
- If action leads to a punishment, avoid that action.

Reinforcement learning: basic approaches

Exploration vs Explotation

Exploit: use your learning result to maximise expected utility now, according to your learned model

Explore: choose an action that will help you improve your model

- How to explore
  - choose a random action
  - choose an action you haven't chosen yet
  - choose and action that will take you to unexplored states
- How to exploit: follow policy
- When to explore

Model-based reinforcement learning

- Learn MDP
- Solve the MDP to determine optimal policy
- Treat the difference between expected / actual reward as an error signal

Model-free reinforcement learning

Don't learn a model, learn the (utility) function directly: Q-learning, TD learning

Next lecture

Game theory and adversarial search!