Offline Planning in MDPs

Tom Silver

Robot Planning Meets Machine Learning

Princeton University

Fall 2025

Building Toward MDPs

Markov Chains

Markov chain: sequence of random variables $S_0, S_1, S_2, \ldots$, with the same domain $\mathcal{S}$ s.t. **Markov property** holds:

for all $t \geq 0$. $P(S_{t+1} \mid S_t, S_{t-1}, \ldots) = P(S_{t+1} \mid S_t)$.

"The future is independent of the past given the present."

Markov Chains

Each S_t represents a **state**

Markov chain: sequence of random variables $S_0, S_1, S_2, \ldots$, with the same domain $\mathcal{S}$ s.t. **Markov property** holds:

The **state space**

for all $t \geq 0$. $P(S_{t+1} \mid S_t, S_{t-1}, \ldots) = P(S_{t+1} \mid S_t)$.

"The future is independent of the past given the present."

Markov Chains

Markov chain: sequence of random variables $S_0, S_1, S_2, \dots$, with the same domain $\mathcal{S}$ s.t. **Markov property** holds:

$$\textit{for all } t \geq 0. \quad P(S_{t+1} \mid S_t, S_{t-1}, \dots) = P(S_{t+1} \mid S_t).$$

Assuming discrete time

"The future is independent of the past given the present."

Markov Chains

Transition distribution for time t: $P_t(S_{t+1} \mid S_t)$

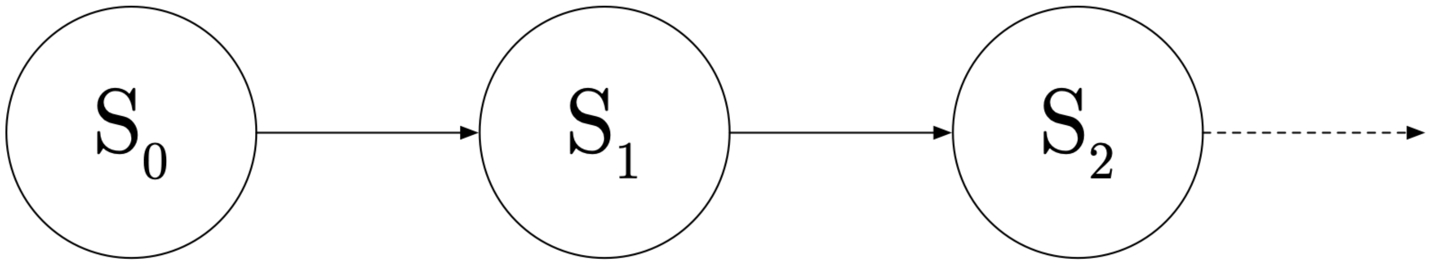
Stationary Markov chain: *for all* $t, t', P_t = P_{t'}$.

a.k.a. time-homogeneous

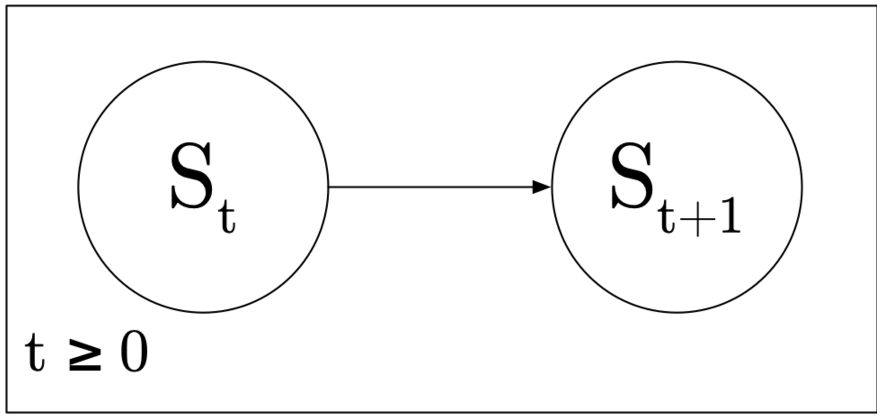
"The transition distribution doesn't change over time."

Notation for stationary MCs omits subscript: $P(S_{t+1} \mid S_t)$

Markov Chain PGM



Graphical model for a Markov chain



Graphical model "plate notation"
for a time-homogenous Markov chain

Example

How would we represent this scenario as a Markov Chain?

Markov Reward Processes

Markov reward process: Markov chain + reward function

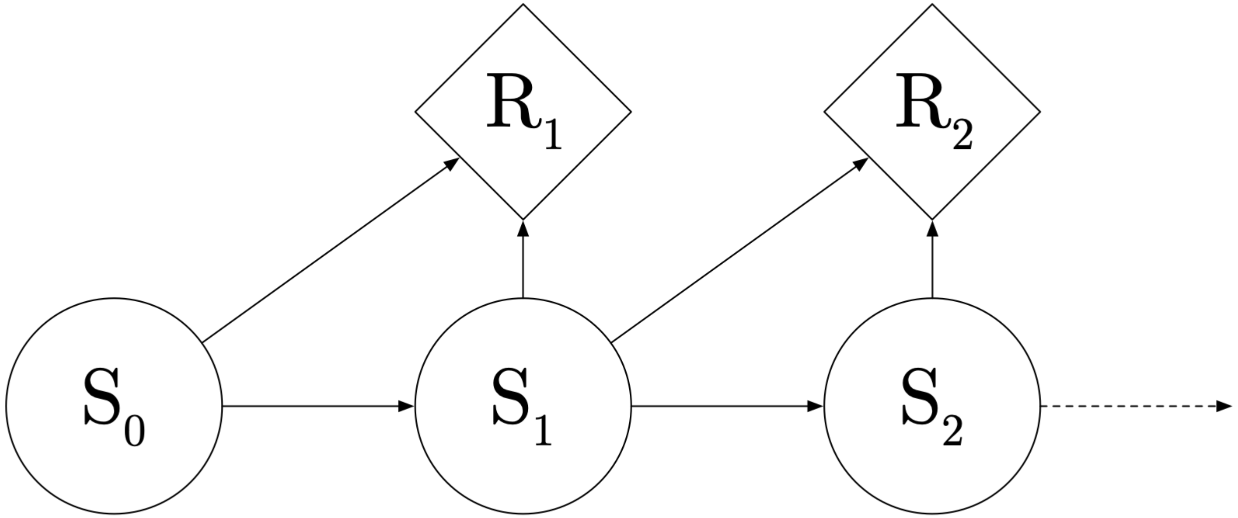
Reward function: $R: \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ (higher is better)

$\mathrm{R}(s_t, s_{t+1}) \mapsto r$ is the *scalar* **reward**

$R(S_t, S_{t+1})$ is a *random variable*

MRP PGM
(Influence Diagram)

Influence diagrams are an extension of Bayes nets that include "reward nodes" (diamonds)



Influence diagram for a Markov Reward Process (MRP)

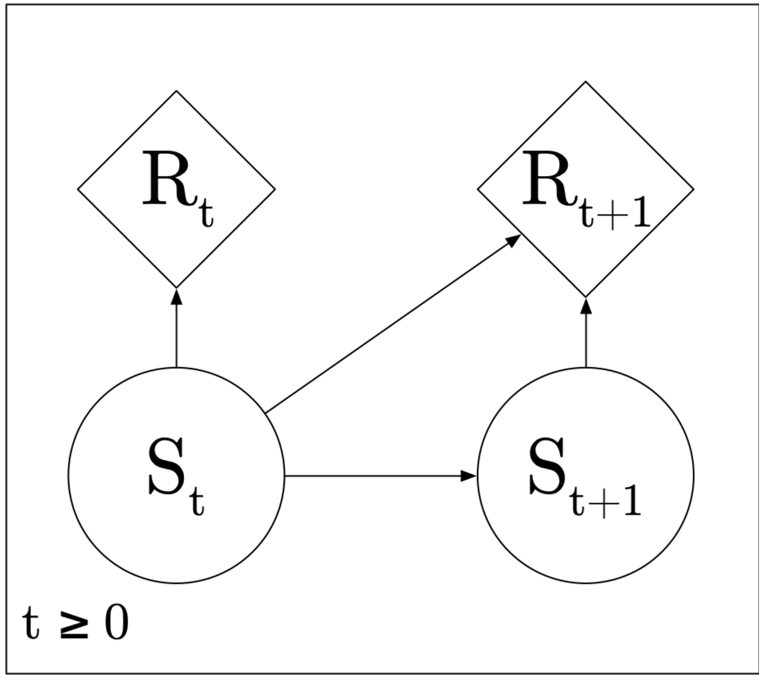


Plate notation for time-homogenous MRP

Time Horizons

- **Finite horizon**: $S_0, S_1, \ldots, S_H$.
- **Infinite horizon:** $S_0, S_1, \ldots$

H is the **horizon**

Time Horizons

- **Finite horizon**: $S_0, S_1, \ldots, S_H$.
- **Infinite horizon:** $S_0, S_1, \ldots$
- **Indefinite horizon:**
  - Let $\mathcal{D} \subset \mathcal{S}$ be a set of **done states**.
  - The process terminates whenever a state in $\mathcal{D}$ is encountered.

a.k.a. sink states, terminal states, absorbing states

Time Horizons

- **Finite horizon**: $S_0, S_1, \dots, S_H$.
- **Infinite horizon:** $S_0, S_1, \dots$
- **Indefinite horizon:**
  - Let $\mathcal{D} \subset \mathcal{S}$ be a set of **done states**.
  - The process terminates whenever a state in $\mathcal{D}$ is encountered.

For indefinite horizon, usually assume that we will reach a done state with probability 1.

Example

How would we represent this scenario as an MRP?


MRP 1
MRP 2

Utilities

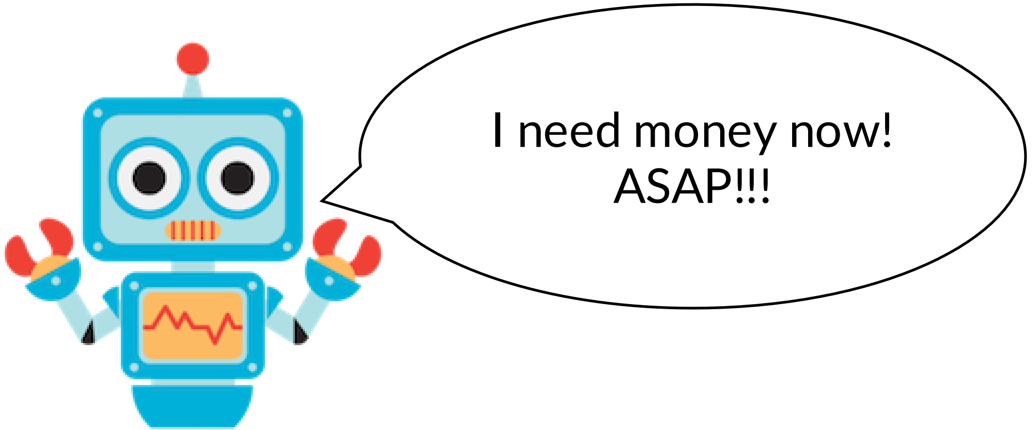
A **utility** for an MRP is a function $U(r_1, r_2, \ldots,) \mapsto u \in \mathbb{R}$.

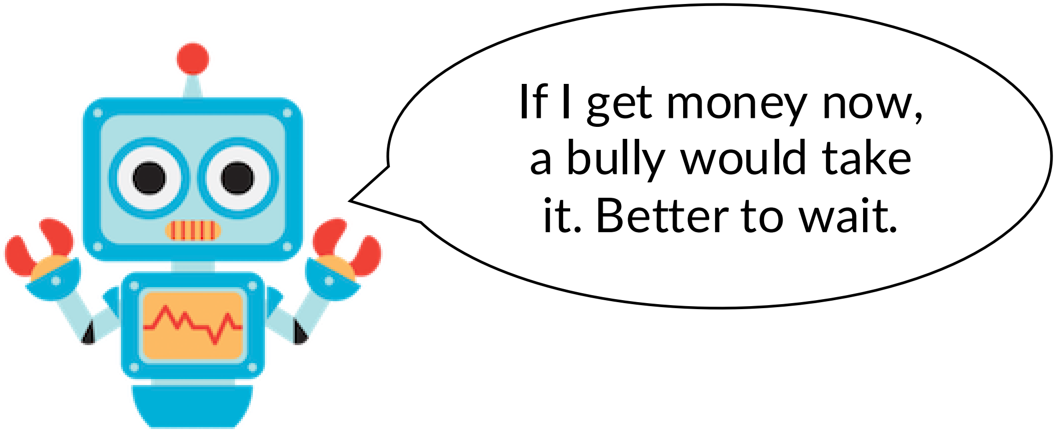
Utility is "what we really want to maximize."

Utilities

A **utility** for an MRP is a function $U(r_1, r_2, \ldots,) \mapsto u \in \mathbb{R}$.

Utility is "what we really want to maximize."



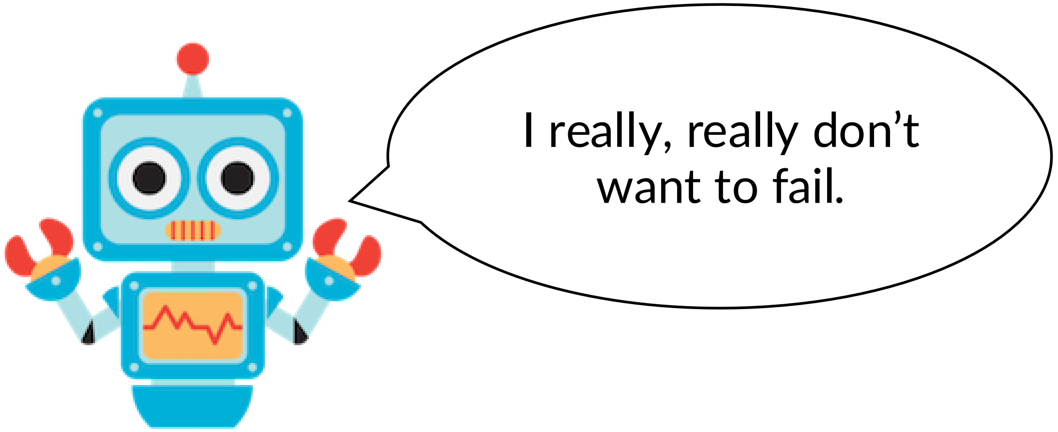


What would each agent's utility look like?

Utilities

A **utility** for an MRP is a function $U(r_1, r_2, \ldots,) \mapsto u \in \mathbb{R}$.

Utility is "what we really want to maximize."



What would each agent's utility look like?

Utilities

A **utility** for an MRP is a function $U(r_1, r_2, \dots,) \mapsto u \in \mathbb{R}$.

Utility is "what we really want to maximize."

Maximum expected utility (MEU) principle: given two MRPs, a rational agent should prefer the one that has larger *expected utility*, where the expectation is over (state, reward) trajectories.

Utilities

A **utility** for an MRP is a function $U(r_1, r_2, \dots,) \mapsto u \in \mathbb{R}$.

Utility is "what we really want to maximize."

Maximum expected utility (MEU) principle: given two MRPs, a rational agent should prefer the one that has larger *expected utility*, where the expectation is over (state, reward) trajectories.

Note: for the moment, we're assuming some distribution over initial states.

Utilities

A **utility** for an MRP is a function $U(r_1, r_2, \dots,) \mapsto u \in \mathbb{R}$.

Utility is "what we really want to maximize."

Maximum expected utility (MEU) principle: given two MRPs, a rational agent should prefer the one that has larger *expected utility*, where the expectation is over (state, reward) trajectories.

Seems reasonable, but is this the only option?

Preferences, Axioms of Utility Theory

- Suppose an agent has **preferences**:
  - $A > B$ (the agent prefers A over B)
  - $A \sim B$ (the agent is indifferent)
  - $A \geq B$ (the agent prefers A over B or is indifferent)

What are A and B here?
Formally: *lotteries*.
For our purposes: MRPs.

Preferences, Axioms of Utility Theory

- Suppose an agent has **preferences**:
  - $A > B$ (the agent prefers A over B)
  - $A \sim B$ (the agent is indifferent)
  - $A \geq B$ (the agent prefers A over B or is indifferent)
- Axioms of utility theory:
  - *Orderability*: exactly one of $A > B$, $B > A$, or $A \sim B$ holds for all A, B.
  - *Transitivity*: if $A > B$ and $B > C$ then $A > C$.
  - Four other more technical ones using *lottery* definition: *continuity*, *substitutability*, *monotonicity*, *decomposability*.

Preferences, Axioms of Utility Theory

Theorem (von Neumann & Morgenstern, 1944): Axioms of utility $\Rightarrow$ maximum expected utility principle.

So, if we accept the axioms, then MEU is what we need.

Our Agent's Utility vs Our Own

Value Alignment Problem

- Need to be very careful & thoughtful about utility definitions!
- Difficult to capture all societal norms
- Famous example: paperclip maximizer (Bostrom 2003)
- Right: another example (OpenAI 2016)

Utilities

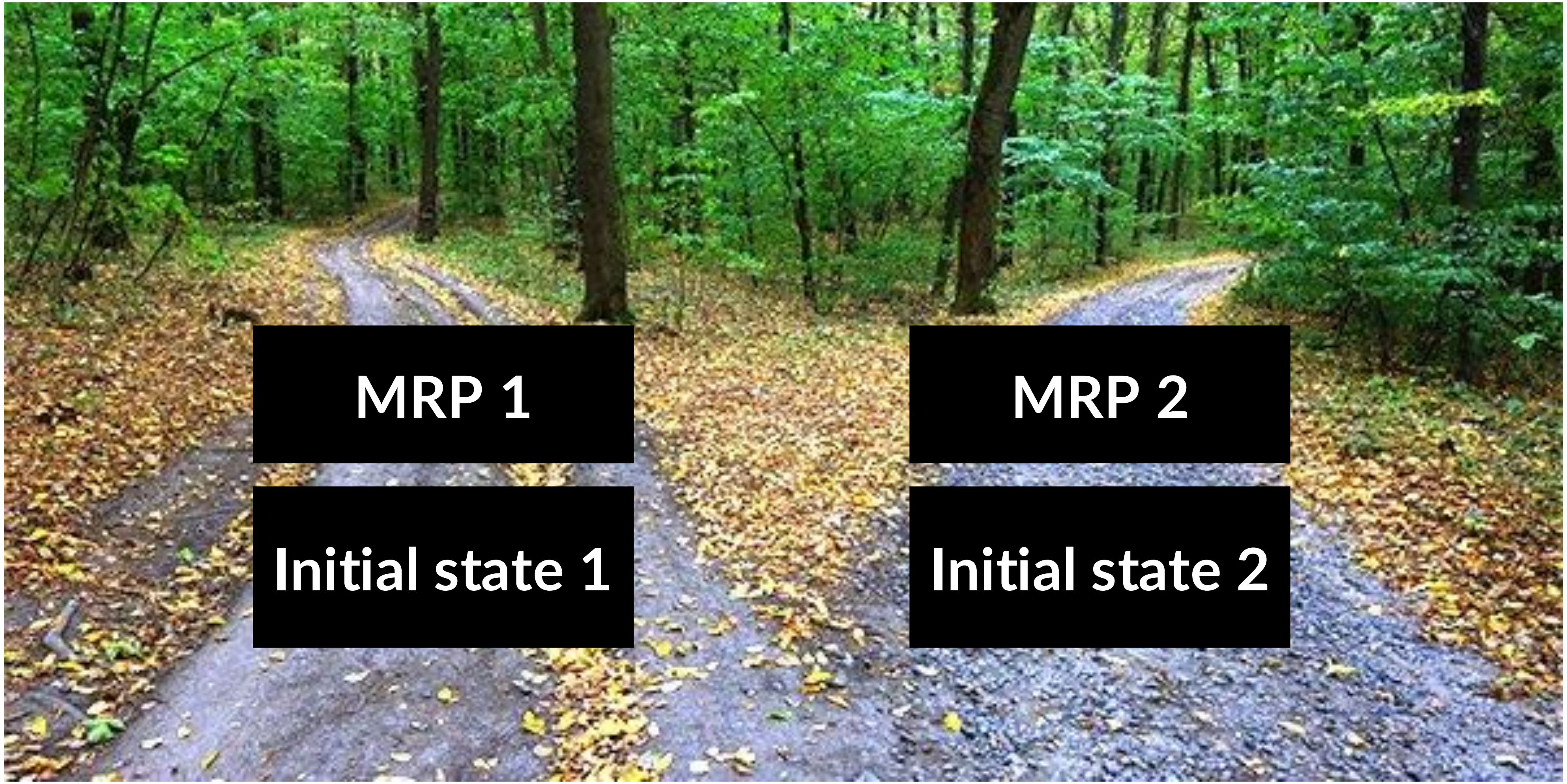
A **utility** for an MRP is a function $U(r_1, r_2, \ldots,) \mapsto u \in \mathbb{R}$.

Utility is "what we really want to maximize."

Maximum expected utility (MEU) principle: given two MRPs, a rational agent should prefer the one that has larger *expected utility*, where the expectation is over (state, reward) trajectories.

But MRPs don't have an initial state distribution! Let's revisit…

Note: for the moment, we're assuming some distribution over initial states.


MRP 1
Initial state 1
MRP 2
Initial state 2

Value Functions

The **value function** $V_t \; : \mathcal{S} \to \mathbb{R}$ for an MRP and utility gives the *expected conditional utility* for starting at $S_t = s$:

$$V_t(s) = E_{S_{t+1},\ldots,S_H | S_t = s}[U(R_{t+1}, R_{t+2} \ldots)]$$

Revised MEU for MRPs

Given two MRPs with the same state space $\mathcal{S}$, let V_t^1, V_t^2 be the respective value functions.

Revised MEU: MRP 1 $\geq$ MRP 2 if $\forall\, t, s.\, V_t^1(s) \geq V_t^2(s)$.

"I prefer MRP 1 over MRP2 (or am indifferent) if for any time and state, the expected conditional utility for MRP1 is at least that of MRP2."

Revised MEU for MRPs

Given two MRPs with the same state space $\mathcal{S}$, let V_t^1, V_t^2 be the respective value functions.

Revised MEU: MRP 1 $\geq$ MRP 2 if $\forall\, t, s.\, V_t^1(s) \geq V_t^2(s)$.

How to evaluate in practice?

"I prefer MRP 1 over MRP2 (or am indifferent) if for any time and state, the expected conditional utility for MRP1 is at least that of MRP2."

Additive Utility Functions

We will now focus on *additive* utility functions:

- Finite horizon: $U(r_1, r_2, \ldots, r_H) = \sum r_t$.

Additive Utility Functions

We will now focus on *additive* utility functions:

- Finite horizon: $U(r_1, r_2, \ldots, r_H) = \sum r_t$.
- Infinite or indefinite horizon: $U(r_1, r_2, \ldots) = \sum \gamma^t r_t$

  where $\gamma \in [0, 1]$ is a **temporal discount factor**.

Infinite horizons: $\gamma \in [0, 1)$

Value Functions with Additive Utility

With additive utility, value functions give *expected cumulative rewards*:

$V_t(s) = E_{S_{t+1},\dots,S_H | S_t = s}[R_{t+1} + \cdots + R_H]$ (finite horizon)

Value Functions with Additive Utility

With additive utility, value functions give *expected cumulative rewards*:

$V_t(s) = E_{S_{t+1},\ldots,S_H | S_t = s}[R_{t+1} + \cdots + R_H]$ (finite horizon)

$V(s) = E_{S_{t+1},\ldots | S_t = s}[R_{t+2} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots]$ (infinite horizon)

No need for time subscripts in infinite horizon case

Bellman Equations: Convenient Recursions

Finite Horizon

$$V_t(s) = \sum_{s'} P(s' \mid s)[R(s, s') + V_{t+1}(s')]$$

$$V_H(s) = 0$$

Immediate reward

Future rewards

Infinite Horizon

$$V(s) = \sum_{s'} P(s' \mid s)[R(s, s') + \gamma V(s')]$$

Bellman Equations: Convenient Recursions

Finite Horizon

$$V_t(s) = \sum_{s'} P(s' \mid s)[R(s, s') + V_{t+1}(s')]$$

$$V_H(s) = 0$$

Can solve for V_t using dynamic programming.
Compute "backwards" from V_H.

Bellman Equations: Convenient Recursions

This is a system of linear equations with $|\mathcal{S}|$ unknowns and $|\mathcal{S}|$ equations. Can solve for V using linear algebra ($\approx$ cubic time).

Infinite Horizon

$$\boxed{V(s) = \sum_{s'} P(s' \mid s)[R(s, s') + \gamma V(s')]}$$

Recap

How to choose an MRP?

1. Solve Bellman equations to get value functions.

2. Check preference property: does one value function "dominate" the other, across all states?

Now assuming:
- Shared state space
- Additive utility

Toward *Sequential* Decision-Making

Given a choice between MRPs, we now know how to pick our favorite one.

This is one-time decision making, at the MRP level.

What if we get to make decisions *at each time step*, influencing the distribution over next states?

Markov Decision Processes

Markov decision process (MDP): MRP + actions.

- State space $\mathcal{S}$
- **Action space $\mathcal{A}$**
- Reward function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$
- Transition distribution $P(S_{t+1} \mid A_t, S_t)$

a_t is action at time t
A_t is random variable for action at time t

Markov Decision Processes

Markov decision process (MDP): MRP + actions.

- State space $\mathcal{S}$
- **Action space** $\mathcal{A}$
- Reward function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$

Reward now depends on action

- Transition distribution $P(S_{t+1} \mid A_t, S_t)$

Transitions now depends on action

Markov Decision Processes

Markov decision process (MDP): MRP + actions.

- State space $\mathcal{S}$
- **Action space** $\mathcal{A}$
- Reward function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$
- Transition distribution $P(S_{t+1} \mid A_t, S_t)$

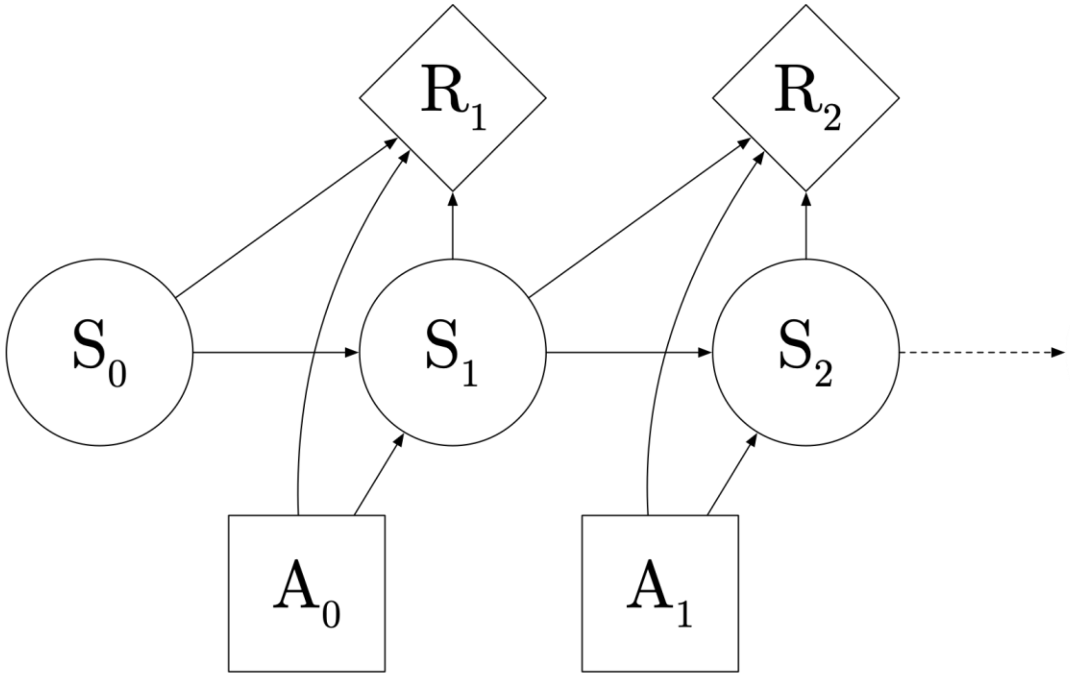
Assumption until we say otherwise:

The state space $\mathcal{S}$ and action space $\mathcal{A}$ are finite.

Not generally true for MDPs.
Just convenient for algorithms.

MDP PGM
(Influence Diagram)

Influence diagrams can also include "decision nodes" (squares)



Influence diagram for a Markov Decision Process (MDP)

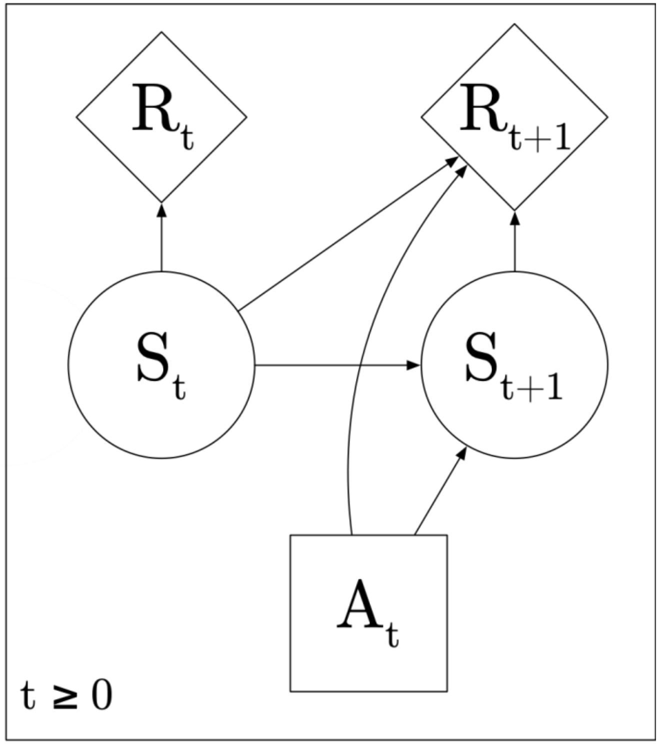
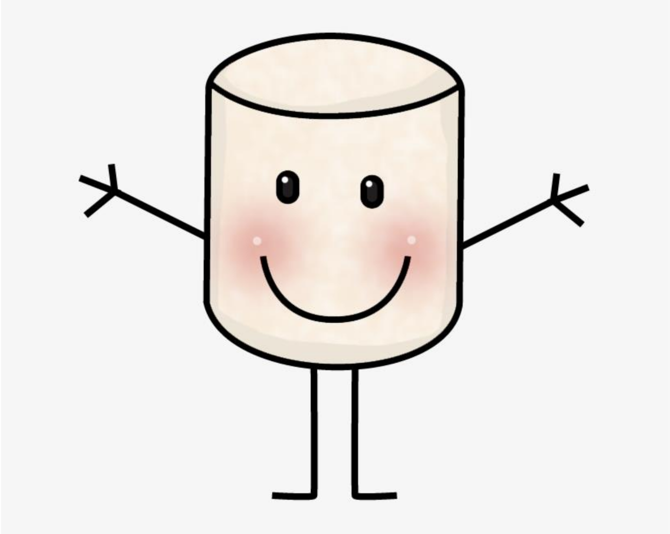


Plate notation for time-homogenous MDP

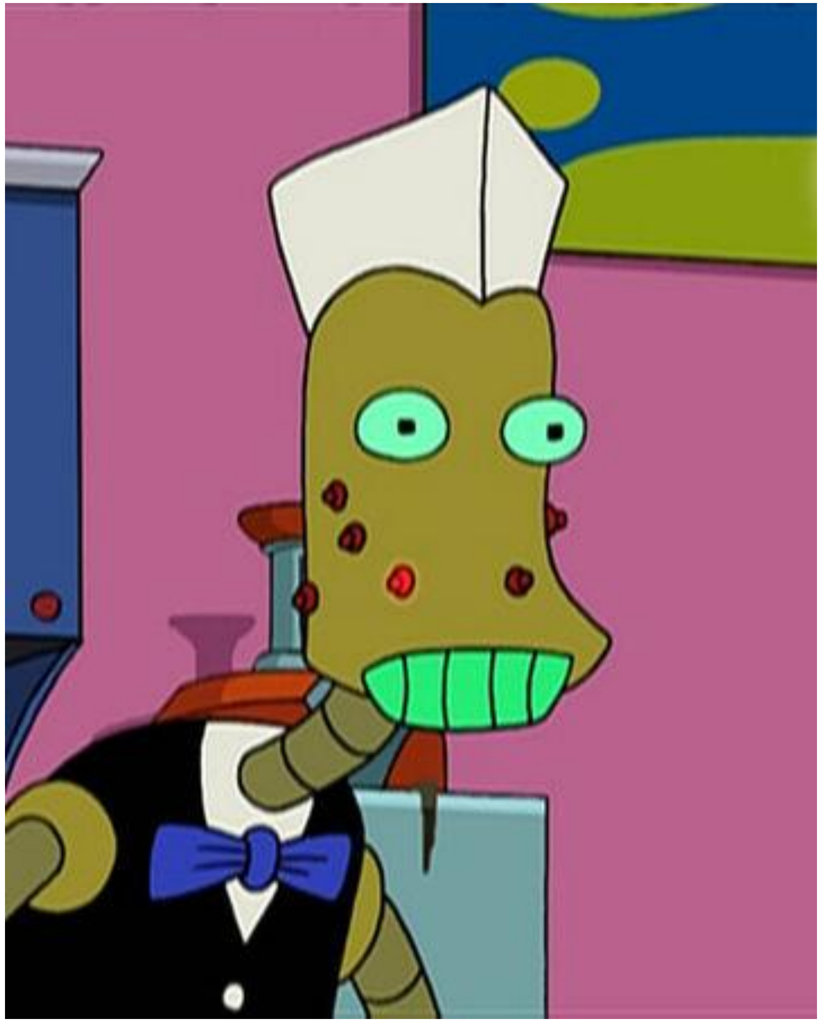
Example: Marshmallows

- **States**: (hunger level, marshmallow remains)
  - Hunger level: 0, 1, 2 (higher is hungrier)
  - Marshmallow remains: True or False
- **Actions**: *eat* marshmallow, or *wait*
- **Horizon:** finite (horizon $H = 4$)
- **Rewards:** Negative hunger level squared (on next state)
- **Transition distribution**:
  - Marshmallow remains updated in obvious way
  - If *wait:*
    - With probability 0.25, hunger level increases by 1
    - Otherwise, hunger level stays the same
  - If *eat* (and marshmallow remains):
    - With probability 1, hunger level set to 0
  - If *eat* (and marshmallow gone):
    - Same as waiting

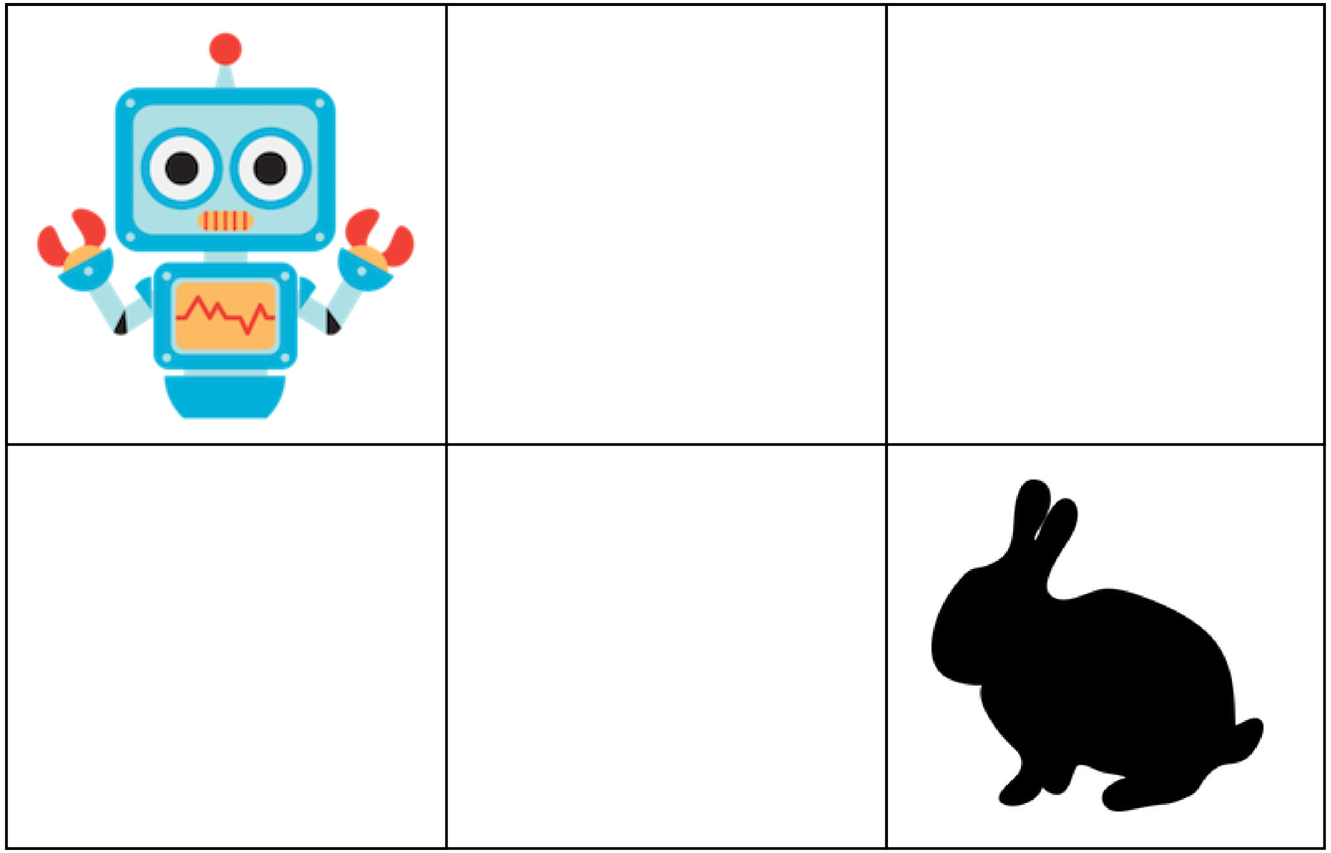


Example: Zits

- **States**: Number of zits on my face: 0, 1, 2, 3, 4
- **Actions**: *apply* zit cream, or just *sleep*
- **Horizon:** infinite (Temporal discount: $\gamma = 0.9$)
- **Rewards:**
  - R(s, *apply*, s') = -(# zits on my face in s') - 1
  - R(s, *sleep*, s') = -(# zits on my face in s')
- **Transition distribution**:
  - If *apply*:
    - With probability 0.8, all zits gone (0)
    - With probability 0.2, all zits grow (4)
  - If *sleep*:
    - With probability 0.4, 1 more zit grows
    - With probability 0.6, 1 zit disappears



Example: Chase



- **States**: (robot pos, rabbit pos)
- **Actions**: move robot up, down, left, right
- **Horizon:** indefinite
  - **Done states**: robot pos = rabbit pos
  - **Temporal discount**: $\gamma = 0.9$
- **Rewards**:
  - +1 for transition that ends in done
  - 0 otherwise
- **Transition distribution**:
  - robot pos is updated deterministically
  - rabbit stays in same place with prob 0.5
  - otherwise jumps to neighboring pos with uniform prob

Policies

A **policy** is a function from states to actions.

Can be **stationary**: $\pi : \mathcal{S} \to \mathcal{A}$

or time-dependent: $\pi_t : \mathcal{S} \to \mathcal{A}$

Policies

A **policy** is a function from states to actions.

Can be **stationary**: $\pi : \mathcal{S} \rightarrow \mathcal{A}$

or time-dependent: $\pi_t : \mathcal{S} \rightarrow \mathcal{A}$

MDP Planning: Find a "good" policy.

What exactly does "good" mean?

Policy + MDP = MRP

- Consider the process of generating states and rewards by following a policy π in an MDP
- This process is an MRP!

Policy + MDP = MRP

- Consider the process of generating states and rewards by following a policy π in an MDP

- This process is an MRP!

From the MDP

- MRP transition distribution: $P(s' \mid s) = P(s' \mid s, \pi(s))$

Policy + MDP = MRP

- Consider the process of generating states and rewards by following a policy π in an MDP

- This process is an MRP!

From the MDP

- MRP transition distribution: $P(s' \mid s) = P(s' \mid s, \pi(s))$

From the MDP

- MRP reward function: $R(s, s') = R(s, \pi(s), s')$

MDP

=

Value Functions for Policies

The **value function** $V_t^\pi: \mathcal{S} \rightarrow \mathbb{R}$ for a policy π in an MDP is the value function for the induced MRP.

In other words, $V_t^\pi(s)$ gives the expected conditional utility for starting at $S_t = s$ and *following* π.

Value Functions for Policies

The **value function** $V_t^\pi : \mathcal{S} \to \mathbb{R}$ for a policy π in an MDP is the value function for the induced MRP.

In other words, V_t^π gives the expected conditional utility for starting at $S_t = s$ and *following* π.

Policy evaluation:
computing V^π given π.

Bellman Equations: Convenient Recursions

Finite Horizon

$$V_t^{\pi}(s) = \sum_{s'} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + V_{t+1}^{\pi}(s')]$$

$$V_H^{\pi}(s) = 0$$

Infinite Horizon

$$V^{\pi}(s) = \sum_{s'} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Bellman Equations: Convenient Recursions

Finite Horizon

$$V_t^{\pi}(s) = \sum_{s'} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + V_{t+1}^{\pi}(s')]$$

$$V_H^{\pi}(s) = 0$$

Can solve for V_t using dynamic programming.
Compute "backwards" from V_H.

Bellman Equations: Convenient Recursions

This is a system of linear equations with $|\mathcal{S}|$ unknowns and $|\mathcal{S}|$ equations. Can solve for V using linear algebra ($\approx$ cubic time).

Infinite Horizon

$$\boxed{\underbrace{V^{\pi}(s) = \sum_{s'} P(s' \mid s, \pi(s))[R(s, \pi(s), s') + \gamma V^{\pi}(s')]}}$$

Planning: Finding an Optimal Policy

Optimal value function: $V^*(s) = \max_{\pi} V^\pi(s)$

Optimal policy: π^* s.t. $\forall s \in \mathcal{S}$. $V^{\pi^*}(s) = V^*(s)$.

"A policy is optimal if it always takes an action that leads to maximum expected utility."

Stupidest Possible Algorithm (SPA) for MDP Planning

PolicyEnumeration($\mathcal{S}, \mathcal{A}, P, R, \gamma$)

// Π is set of all possible policies
for $\pi \in \Pi$
    If π is best seen so far, keep it
return Best seen π

Stupidest Possible Algorithm (SPA) for MDP Planning

PolicyEnumeration$(\mathcal{S}, \mathcal{A}, P, R, \gamma)$

```
// Π is set of all possible policies
for π ∈ Π
    If π is best seen so far, keep it
return Best seen π
```

Review: how would we check this?

Policy Iteration: A Less Stupid Algorithm

"Find a policy improvement."

POLICYITERATION($\mathcal{S}, \mathcal{A}, P, R, \gamma$)

// Initialize a policy π arbitrarily.
repeat
    Compute V^π.
    Find $s \in \mathcal{S}, a \in \mathcal{S}$ s.t. $E_{S_{t+1} | S_t = s, \mathbf{A_t} = \mathbf{a}}[V^\pi(S_{t+1})] > E_{S_{t+1} | S_t = s, \mathbf{A_t} = \pi(\mathbf{s})}[V^\pi(S_{t+1})]$.
    If none exist, **return** π.
    Otherwise, update $\pi(s) = a$.

Policy Iteration: A Less Stupid Algorithm

PolicyIteration$(\mathcal{S}, \mathcal{A}, P, R, \gamma)$

```
// Initialize a policy π arbitrarily.
repeat
    Compute V^π.
    Find s ∈ S, a ∈ S s.t. E_{S_{t+1}|S_t=s, A_t=a}[V^π(S_{t+1})] > E_{S_{t+1}|S_t=s, A_t=π(s)}[V^π(S_{t+1})].
    If none exist, return π.
    Otherwise, update π(s) = a.
```

Guaranteed to converge to an optimal policy.

Policy Iteration: A Less Stupid Algorithm

$\textsc{PolicyIteration}(\mathcal{S}, \mathcal{A}, P, R, \gamma)$

// Initialize a policy π arbitrarily.
repeat
  Compute V^π.
  Find $s \in \mathcal{S}, a \in \mathcal{S}$ s.t. $E_{S_{t+1} | S_t = s, \mathbf{A_t} = \mathbf{a}}[V^\pi(S_{t+1})] > E_{S_{t+1} | S_t = s, \mathbf{A_t} = \pi(\mathbf{s})}[V^\pi(S_{t+1})]$.
  If none exist, **return** π.
  Otherwise, update $\pi(s) = a$.

This is ugly, let's refactor

Action-Value (Q) Functions

The **action-value function** $Q_t^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ gives the *expected cumulative rewards* for starting at $S_t = s$, taking action $A_t = a$, and then following π:

$$Q_t^\pi(s, a) = E_{S_{t+1} | S_t = s,\, A_t = a} \left[R_t + V_{t+1}^\pi(S_{t+1}) \right]$$

Policy Iteration: Refactored

PolicyIteration($\mathcal{S}, \mathcal{A}, P, R, \gamma$)

// Initialize a policy π arbitrarily.
repeat
    Compute V^π.
    Find $s \in \mathcal{S}, a \in \mathcal{S}$ s.t. $Q^\pi(s, a) > Q^\pi(s, \pi(s))$.
    If none exist, **return** π.
    Otherwise, update $\pi(s) = a$.

Avoiding Policy Evaluation

- Policy iteration requires evaluating the Bellman equations in the inner loop, which can be expensive

- Rather than keeping track of a policy, what if we compute an *optimal value function directly*?

- Once we have the optimal value function, we can compute a corresponding optimal policy at the end

Value Function → Policy

Given an action-value function Q, a **greedy policy** is:

$$\pi(s) = \text{argmax}_a \, Q(s, a)$$

Value Function → Policy

Given an action-value function Q, a **greedy policy** is:

$$\pi(s) = \text{argmax}_a \, Q(s, a)$$

If we have an optimal action-value function Q^*, the greedy policy is an optimal policy:

$$\pi^*(s) = \text{argmax}_a \, Q^*(s, a)$$

Value Function → Policy

Given an action-value function Q, a **greedy policy** is:

$$\pi(s) = \text{argmax}_a\, Q(s, a)$$

If we have an optimal action-value function Q^*, the greedy policy is an optimal policy:

$$\pi^*(s) = \text{argmax}_a\, Q^*(s, a)$$

Are greedy policies unique?

Value Function → Policy

Given an action-value function Q, a **greedy policy** is:

$$\pi(s) = \text{argmax}_a \, Q(s, a)$$

If we have an optimal action-value function Q^*, the greedy policy is an optimal policy:

$$\pi^*(s) = \text{argmax}_a \, Q^*(s, a)$$

So, how can we directly compute V^* / Q^*?

Bellman Equations: Convenient Recursions

Finite Horizon

$$V_t^*(s) = \max_a \sum_{s'} P(s' \mid s, a)[R(s, a, s') + V_{t+1}^*(s')]$$

$$V_H^*(s) = 0$$

Bellman Equations: Convenient Recursions

Finite Horizon

$$V_t^*(s) = \max_a \sum_{s'} P(s' \mid s, a)[R(s, a, s') + V_{t+1}^*(s')]$$

$$V_H^*(s) = 0$$

Can solve for V_t^* using dynamic programming.
Compute "backwards" from V_H^*.

Compute Optimal Value Functions: **Finite Horizon**

ComputeFiniteHorizonValueFunction($\mathcal{S}, \mathcal{A}, P, R, H$)

// Represent values as dictionary `V[t][s]` $= V_t^*(s)$.
`V` $=$ `dict()`
// Base case: final values are 0
for each `s` $\in \mathcal{S}$
    `V[H][s]` $=$ `0`
// Recursive step: compute backwards in time
for `t` $= H-1, H-2, \ldots, 0$
    for each `s` $\in \mathcal{S}$
        `V[t][s]` $= \max_{\mathtt{a} \in \mathcal{A}} \sum_{\mathtt{s'} \in \mathcal{S}} P(\mathtt{s'} \mid \mathtt{s}, \mathtt{a})[R(\mathtt{s}, \mathtt{a}, \mathtt{s'}) + \mathtt{V[t+1][s']}]$
return `V`

Compute Optimal Value Functions: **Finite Horizon**

$\textsc{ComputeFiniteHorizonValueFunction}(\mathcal{S}, \mathcal{A}, P, R, H)$

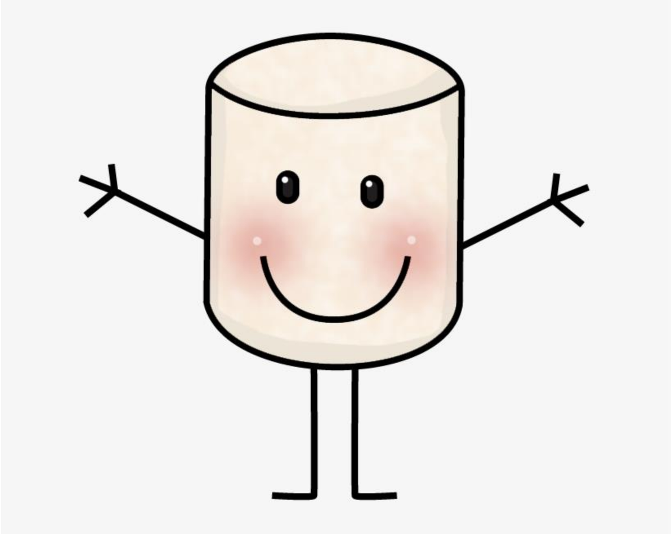
// Represent values as dictionary $\texttt{V[t][s]} = V_t^*(s)$.
$\texttt{V} = \texttt{dict()}$
// Base case: final values are 0
for each $\texttt{s} \in \mathcal{S}$
    $\texttt{V[H][s]} = \texttt{0}$
// Recursive step: compute backwards in time
for $\texttt{t} = H-1, H-2, \ldots, 0$
    for each $\texttt{s} \in \mathcal{S}$
        $\texttt{V[t][s]} = \max_{\texttt{a} \in \mathcal{A}} \sum_{\texttt{s'} \in \mathcal{S}} P(\texttt{s'} \mid \texttt{s}, \texttt{a})[R(\texttt{s}, \texttt{a}, \texttt{s'}) + \texttt{V[t+1][s']}]$
return $\texttt{V}$

Example of **dynamic programming**

What's the asymptotic complexity?

Example: Marshmallows

- **States**: (hunger level, marshmallow remains)
  - Hunger level: 0, 1, 2 (higher is hungrier)
  - Marshmallow remains: True or False
- **Actions**: *eat* marshmallow, or *wait*
- **Horizon:** finite (horizon $H = 4$)
- **Rewards:** Negative hunger level squared (on next state)
- **Transition distribution**:
  - Marshmallow remains updated in obvious way
  - If *wait:*
    - With probability 0.25, hunger level increases by 1
    - Otherwise, hunger level stays the same
  - If *eat* (and marshmallow remains):
    - With probability 1, hunger level set to 0
  - If *eat* (and marshmallow gone):
    - Same as waiting



Initialization (H=4)

- V[4][0T] = 0, V[4][1T] = 0, V[4][2T] = 0, V[4][0F] = 0, V[4][1F] = 0, V[4][2F] = 0

Initialization (H=4)

- V[4][0T] = 0, V[4][1T] = 0, V[4][2T] = 0, V[4][0F] = 0, V[4][1F] = 0, V[4][2F] = 0

Iteration (H=3)

- V[3][0T] = max(

  … // Eat

  … // Wait

)

Initialization (H=4)

- V[4][0T] = 0, V[4][1T] = 0, V[4][2T] = 0, V[4][0F] = 0, V[4][1F] = 0, V[4][2F] = 0

Iteration (H=3)

- V[3][0T] = max(

$$\underbrace{\text{P(0F | E, 0T)(R(0T, E, 0F) + V[4][0F])}}_{\text{Eat} \rightarrow \text{0F}}$$

Zero-prob transitions not written

... // Wait

)

Initialization (H=4)

- V[4][0T] = 0, V[4][1T] = 0, V[4][2T] = 0, V[4][0F] = 0, V[4][1F] = 0, V[4][2F] = 0

Iteration (H=3)

- V[3][0T] = max(

$$\underbrace{P(0F \mid E, 0T)(R(0T, E, 0F) + V[4][0F])}_{\text{Eat} \rightarrow 0F}$$

$$\underbrace{P(0T \mid W, 0T)(R(0T, W, 0T) + V[4][0T])}_{\text{Wait} \rightarrow 0T} + \underbrace{P(1T \mid W, 0T)(R(0T, W, 1T) + V[4][1T])}_{\text{Wait} \rightarrow 1T}$$

)

Initialization (H=4)

- V[4][0T] = 0, V[4][1T] = 0, V[4][2T] = 0, V[4][0F] = 0, V[4][1F] = 0, V[4][2F] = 0

Iteration (H=3)

- V[3][0T] = max(

P(0F | E, 0T)(R(0T, E, 0F) + V[4][0F])

P(0T | W, 0T)(R(0T, W, 0T) + V[4][0T]) + P(1T | W, 0T)(R(0T, W, 1T) + V[4][1T])

)

= max(

1 * (0 + 0)
0.75 * (0 + 0) + 0.25 * (-1 + 0)

) = 0

…

Bellman Backups

The meat of the value function update was this:

$$\texttt{V[t][s]} = \max_{\texttt{a} \in \mathcal{A}} \sum_{\texttt{s'} \in \mathcal{S}} P(\texttt{s'} \mid \texttt{s}, \texttt{a})[R(\texttt{s}, \texttt{a}, \texttt{s'}) + \texttt{V[t+1][s']}]$$

Important enough that we will give it a name: **Bellman backup**

Bellman, because of the Bellman equation.

Backup, because we're looking one step ahead and "backing up".

Compute Optimal Value Functions: **Finite Horizon**

$\textsc{ComputeFiniteHorizonValueFunction}(\mathcal{S}, \mathcal{A}, P, R, H)$

// Represent values as dictionary $\texttt{V[t][s]} = V_t^*(s)$.
$\texttt{V} = \texttt{dict()}$
// Base case: final values are 0
for each $\texttt{s} \in \mathcal{S}$
    $\texttt{V[H][s]} = \texttt{0}$
// Recursive step: compute backwards in time
for $\texttt{t} = H - 1, H - 2, \ldots, 0$
    for each $\texttt{s} \in \mathcal{S}$
        $\texttt{V[t][s]} = \textsc{BellmanBackup}(s, \texttt{V}, \mathcal{S}, \mathcal{A}, P, R, t)$
return $\texttt{V}$

Refactored

Bellman Equations: Convenient Recursions

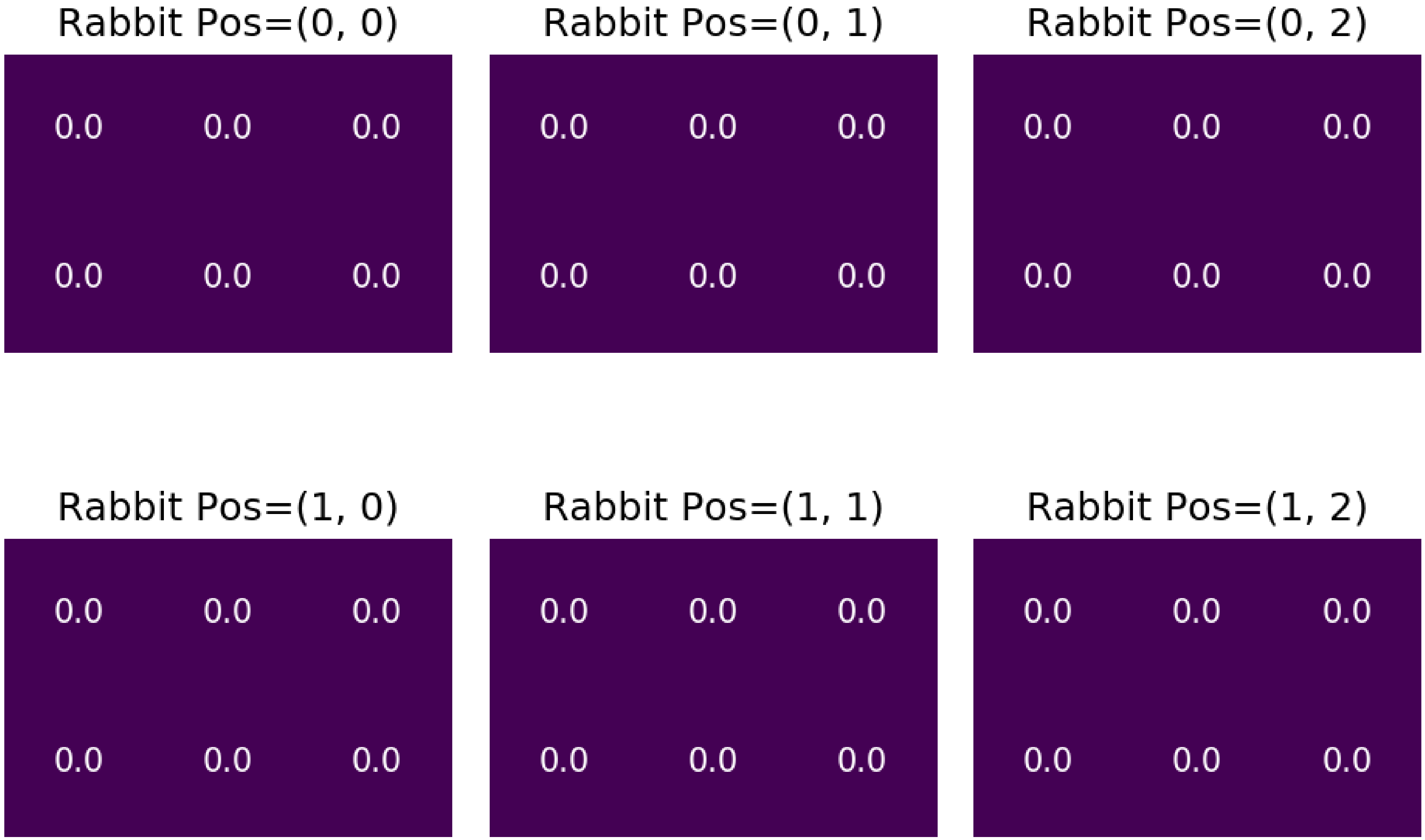
No longer linear! What to do?
Idea: iteratively "plug in" the RHS to update the LHS.

Infinite Horizon

$$\boxed{V^*(s) = \max_{a} \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V^*(s')]}$$

Value Iteration

```
ValueIteration(S, A, P, R, γ)
// Represent values as dictionary V[s] = V*(s), initialized arbitrarily.
while not converged
    // Initialize new value function dictionary
    Vn = dict()
    for each s ∈ S
        Vn[s] = BellmanBackup(s, V, S, A, P, R, γ)
    V = Vn
return V
```


Rabbit Pos=(0, 0)
0.0 0.0 0.0
0.0 0.0 0.0
Rabbit Pos=(0, 1)
0.0 0.0 0.0
0.0 0.0 0.0
Rabbit Pos=(0, 2)
0.0 0.0 0.0
0.0 0.0 0.0
Rabbit Pos=(1, 0)
0.0 0.0 0.0
0.0 0.0 0.0
Rabbit Pos=(1, 1)
0.0 0.0 0.0
0.0 0.0 0.0
Rabbit Pos=(1, 2)
0.0 0.0 0.0
0.0 0.0 0.0

Value Iteration (Continued)

- What's the asymptotic complexity per-iteration?

Value Iteration (Continued)

- What's the asymptotic complexity per-iteration?
  - $O(|\mathcal{S}|^2|\mathcal{A}|)$

Value Iteration (Continued)

- What's the asymptotic complexity per-iteration?
  - $O(|\mathcal{S}|^2|\mathcal{A}|)$
- How should we check convergence?
  - For theoretical guarantees, use $\max_s |V(s) - Vn(s)| < \epsilon$.

Value Iteration (Continued)

- What's the asymptotic complexity per-iteration?
  - $O(|\mathcal{S}|^2|\mathcal{A}|)$
- How should we check convergence?
  - For theoretical guarantees, use $\max_s |V(s) - Vn(s)| < \epsilon$.
- Is this guaranteed to converge to the optimal value function?
  - (Thm) Yes.

Value Iteration (Continued)

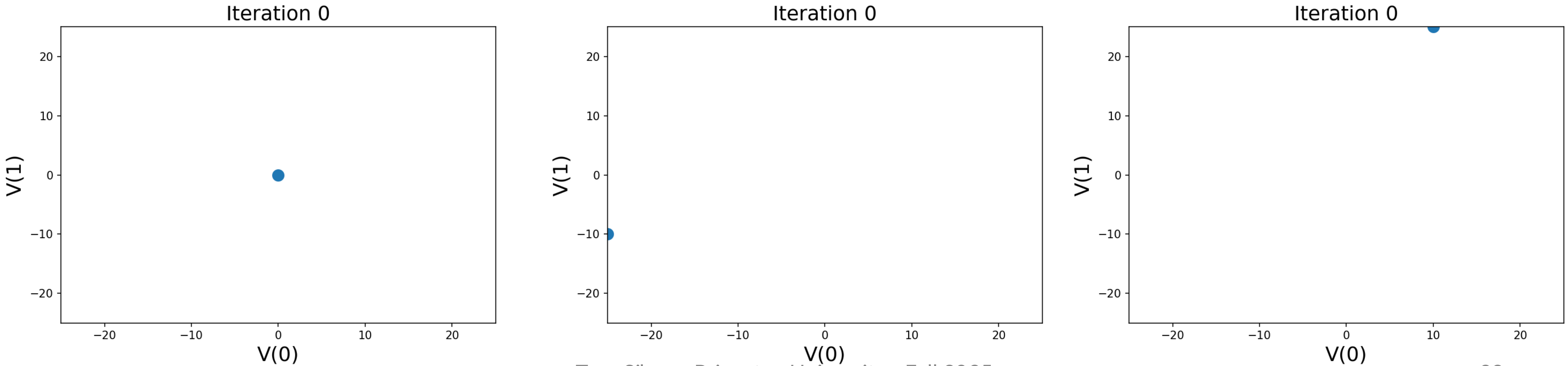
- What's the asymptotic complexity per-iteration?
  - $O(|\mathcal{S}|^2|\mathcal{A}|)$
- How should we check convergence?
  - For theoretical guarantees, use $\max_s |V(s) - Vn(s)| < \epsilon$.
- Is this guaranteed to converge to the optimal value function?
  - (Thm) Yes.
- Does the initialization matter?
  - Asymptotically, no, but it can affect the rate of convergence. (Extreme case: initialize to optimal.)

Why Does Value Iteration Work?

- Each iteration maps one value function V to another, Vn
- We can think about this map as a function $B : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$
  - Here we are representing a value function as a vector

Why Does Value Iteration Work?

- Each iteration maps one value function V to another, Vn
- We can think about this map as a function $B : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$
  - Here we are representing a value function as a vector
- Example: $|\mathcal{S}| = 2$. Three different initializations.



Why Does Value Iteration Work?

- B is a *contraction mapping**
  - There exists some k s.t. for any two inputs v, v',

    $$distance(B(v), B(v')) \le k\ distance(v, v').$$

  That is, all pairs of points get closer after the mapping is applied.

For us, distance is L_∞-norm.

*Short proof: http://www.cs.cmu.edu/afs/cs/academic/class/15780-s16/www/slides/mdps.pdf Slide 19.

Why Does Value Iteration Work?

- B is a *contraction mapping**
  - There exists some k s.t. for any two inputs v, v',

    $distance(B(v), B(v')) \leq k\ distance(v, v')$.

  That is, all pairs of points get closer after the mapping is applied.

- Theorem (Banach fixed point theorem): If B is a contraction mapping, it has a unique fixed point.

- For us, that unique fixed point is the optimal value function.

*Short proof: http://www.cs.cmu.edu/afs/cs/academic/class/15780-s16/www/slides/mdps.pdf Slide 19.

Policy Iteration vs. Value Iteration

- PI typically needs fewer iterations than VI to converge
- However, each iteration of PI requires policy evaluation
- Modified PI (Putterman & Shin, 1978) performs a cheaper approximate policy evaluation. Often the best in practice
- But with modern compute, if you have an MDP that is practically too big for VI, then it’s probably too big for PI and MPI as well, and you need approximate methods

Linear Programming

Compute value function by solving linear program:

$$\text{Minimize } V(s) \text{ for all } s$$
$$\text{subject to } V(s) \geq \sum_{s' \in \mathcal{S}} P(s' \mid s, a)[R(s, a, s') + \gamma V(s')]$$

Less widely used. But, tightest complexity bounds!

And, the basis for some other approximate methods, with connections to other communities.

Next Time

- What if $\mathcal{S}$ is very large?

- If we know our current state, could we leverage it?

- How can we incorporate heuristics?