

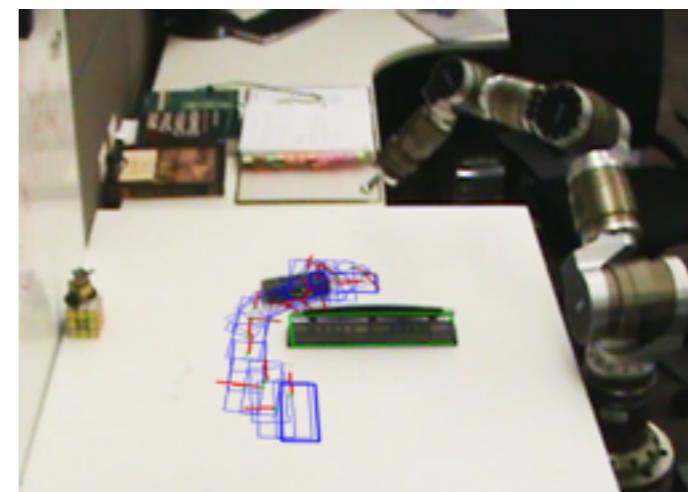
Markov Decision Processes and Reinforcement Learning

An Introduction to Stochastic Planning

Path Planning Assumptions

- Obstacles?
 - Reliable Collision detection (assumes robust perception)
- Transitions
 - Reliable mechanism for moving along path in graph (i.e., a controller)

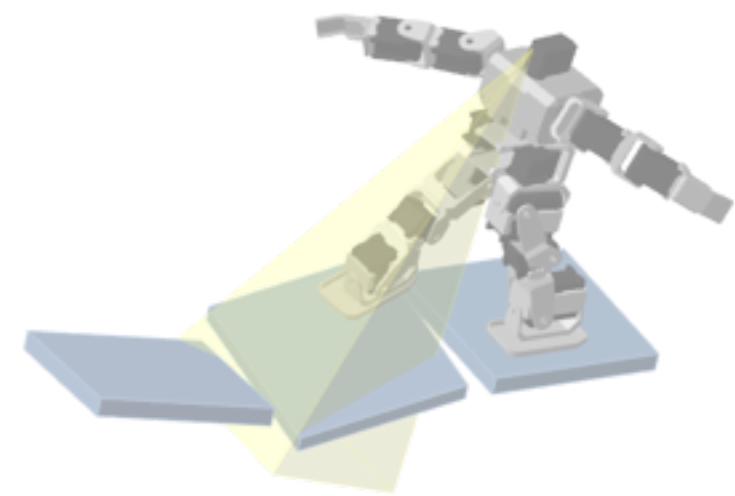
```
move_block(x1,y1,x2,y2)
```



Two Sources of Error

- **State Estimation**

- You don't know exactly where you are
- Sensors have noise
- No complete environment information



- **Action Execution**

- Your actuators don't do what you tell them
- Your system responds differently than you expect
- Friction, gears, air resistance, etc.

Basic Idea: Your model of the world is incorrect!

Markov (Decision) Processes: A New Model for Planning

- Handles both forms of uncertainty in a *statistically principled way*
- Gives us back optimality!
- Of course, I'm talking about (PO)MDPs
 - All this flexibility comes at a cost, as we'll see...
 - Current research is largely about scalability

Handling Non-Deterministic Actions

- **Problem:** we don't know where our actions take us
- **Solution:** start thinking about expected values
 - Weight each outcome by the *probability* of getting there

Formalizing the MDP Model

- **Step 1:** define the core problem representation
 - Considerations?
 1. should represent “rewards” somehow
 2. should represent “state” somehow
 3. should represent “actions” somehow
- next: what if actions aren’t deterministic??

Formalizing the MDP Model

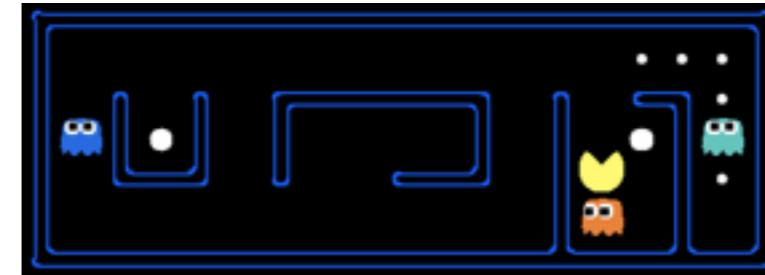
- **Step 2:** How to handle *stochastic* action effects (“transitions”)?
 - replace transition rule with transition distribution

$$T(s, s') = P(s'|s) = \begin{bmatrix} P_{11} & P_{12} & \dots & P_{1n} \\ P_{21} & P_{22} & \dots & P_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ P_{n1} & P_{n2} & \dots & P_{nn} \end{bmatrix}$$

Formalizing the MDP Model

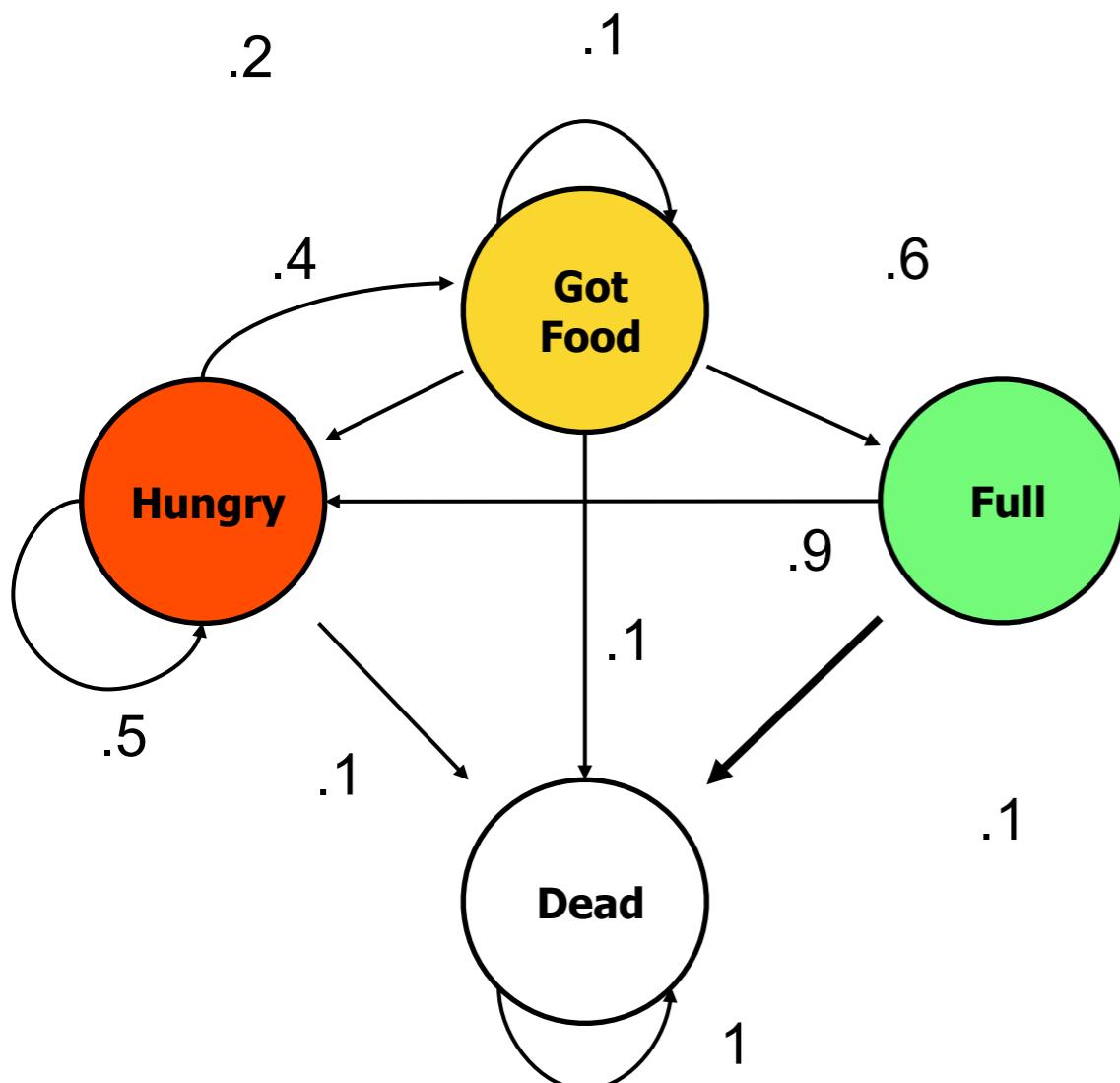
- Overall: $MDP = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma\}$

\mathcal{S} = States
 \mathcal{A} = Actions
 \mathcal{T} = Transition Model
 \mathcal{R} = Rewards



- Pacman states**
 - {all positions of pacman, ghosts, food, & pellets}
- Pacman actions**
 - {N,S,E,W}
- Pacman model**
 - {move directions, die from ghosts, eat food,...}
- Pacman rewards**
 - -1 per step, +10 food, -500 die, +500 win,...

Markov Processes: Caveman's World



States: {H, G, F, D}

Actions: {}

(we'll get back
to this)

Transition Model:

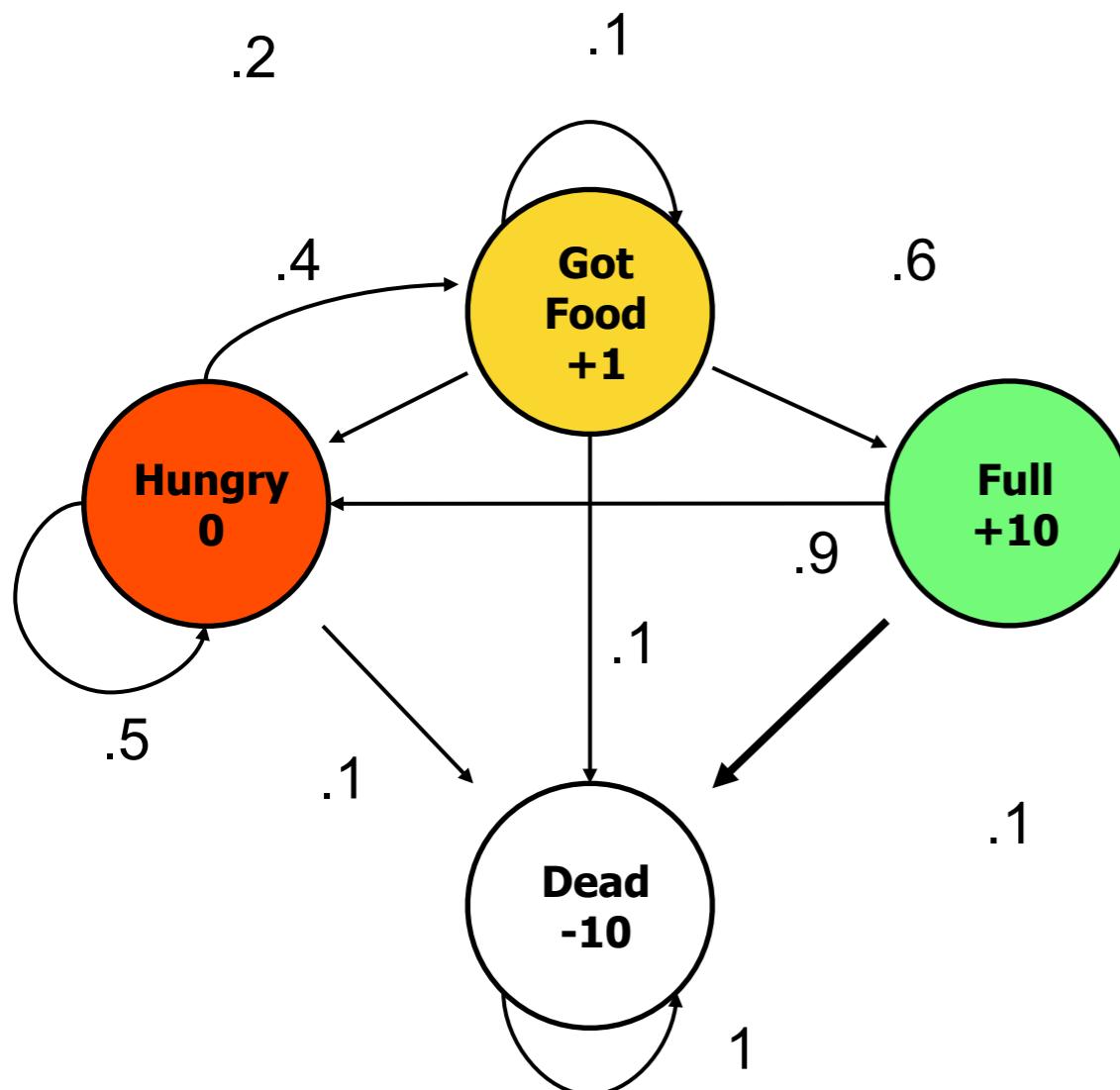
	H	G	F	D	$\Sigma=1.0$
H	0.5	0.4	0.0	0.1	$\Sigma=1.0$
G	0.2	0.1	0.6	0.1	$\Sigma=1.0$
F	0.9	0.0	0.0	0.1	$\Sigma=1.0$
D	0.0	0.0	0.0	1.0	$\Sigma=1.0$

Rewards:

H	G	F	D
0	I	10	-10

just a CPT

Markov Processes: Caveman's World



States: {H, G, F, D}

Actions: {}

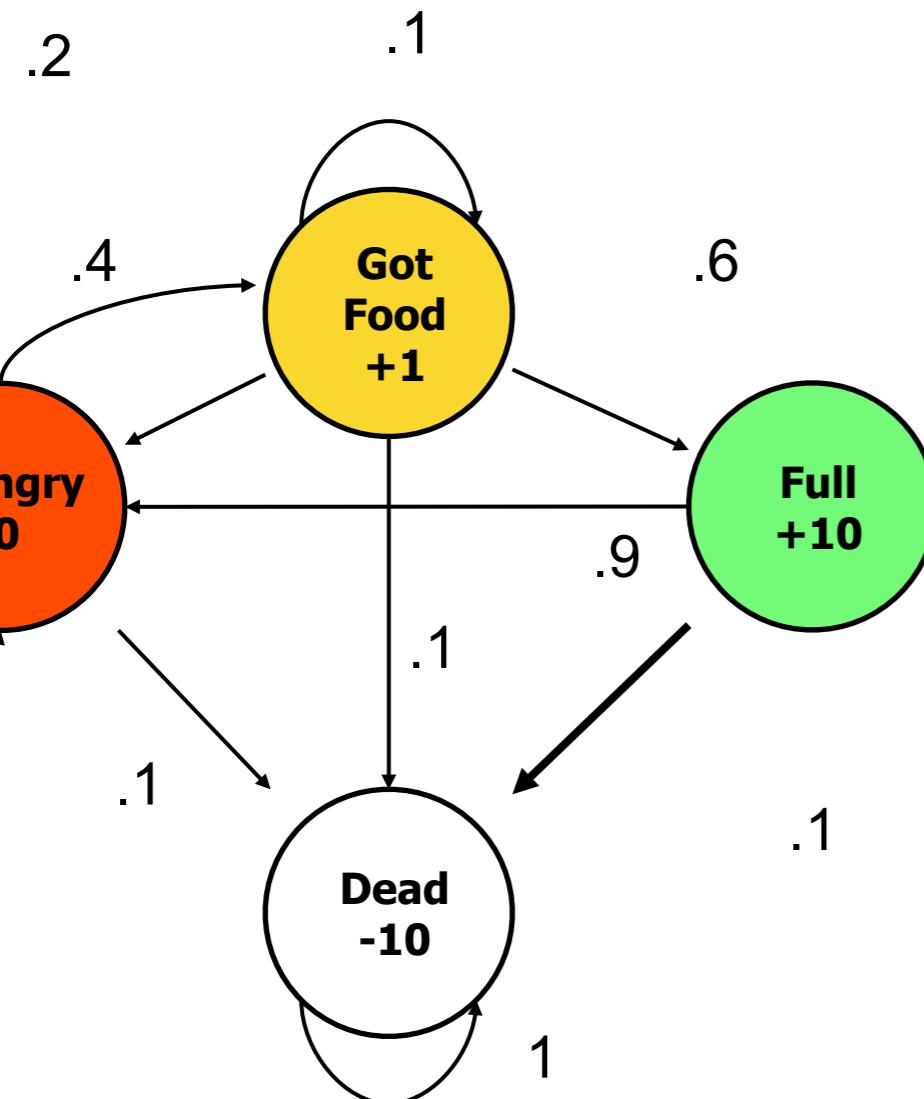
Transition Model:

	H	G	F	D
H	0.5	0.4	0.0	0.1
G	0.2	0.1	0.6	0.1
F	0.9	0.0	0.0	0.1
D	0.0	0.0	0.0	1.0

Rewards:

H	G	F	D
0	1	10	-10

Markov Processes: Caveman's World



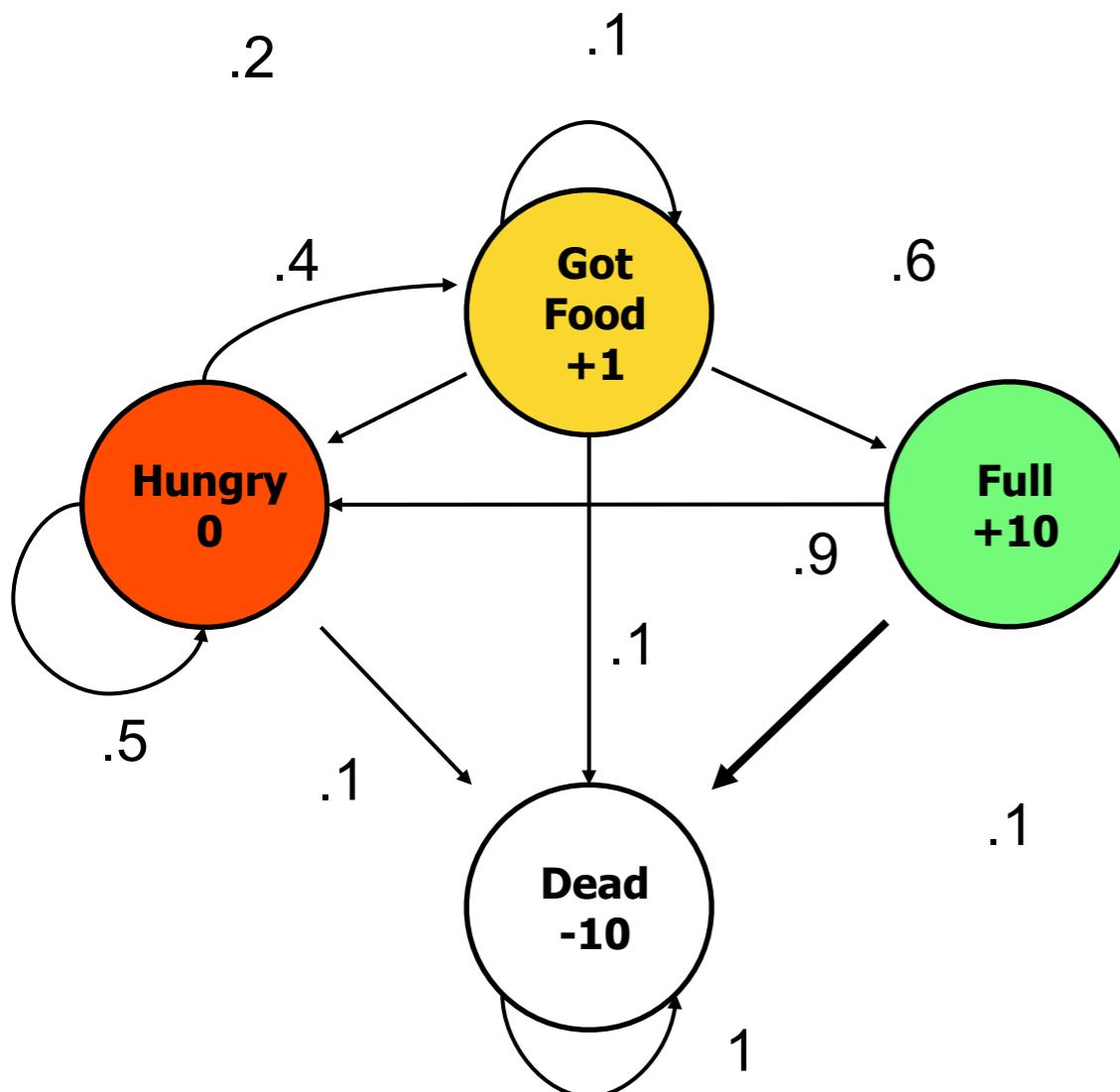
Value

- How good is it to be in a state?
- Sum of **DISCOUNTED** expected rewards:

$$V(s) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

- Reward now is better than later. Why??

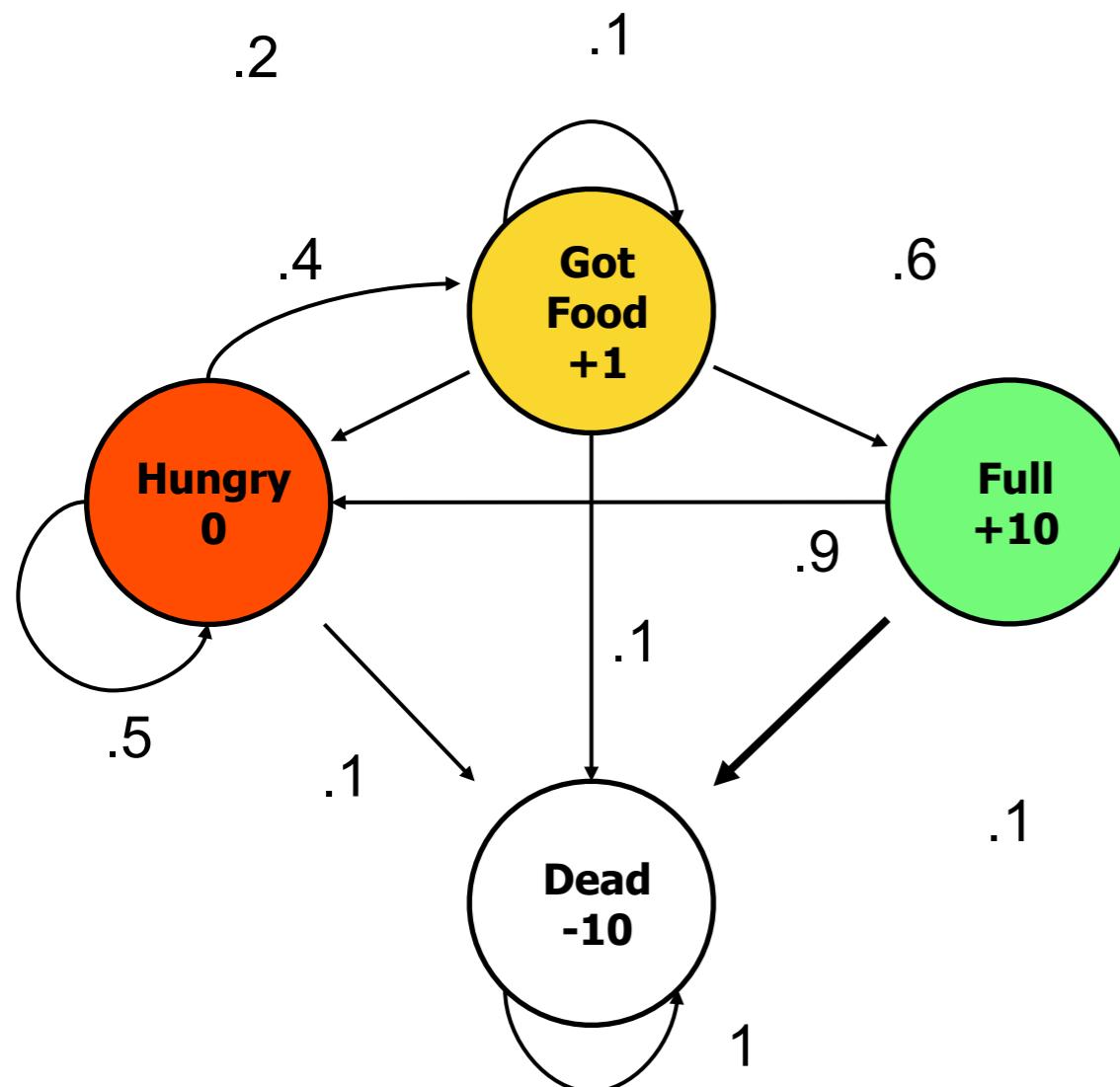
Value Iteration in Caveman's World



- Key idea: Bellman Recursion
- Relates value in current state to expected value of next state

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s)V(s')$$

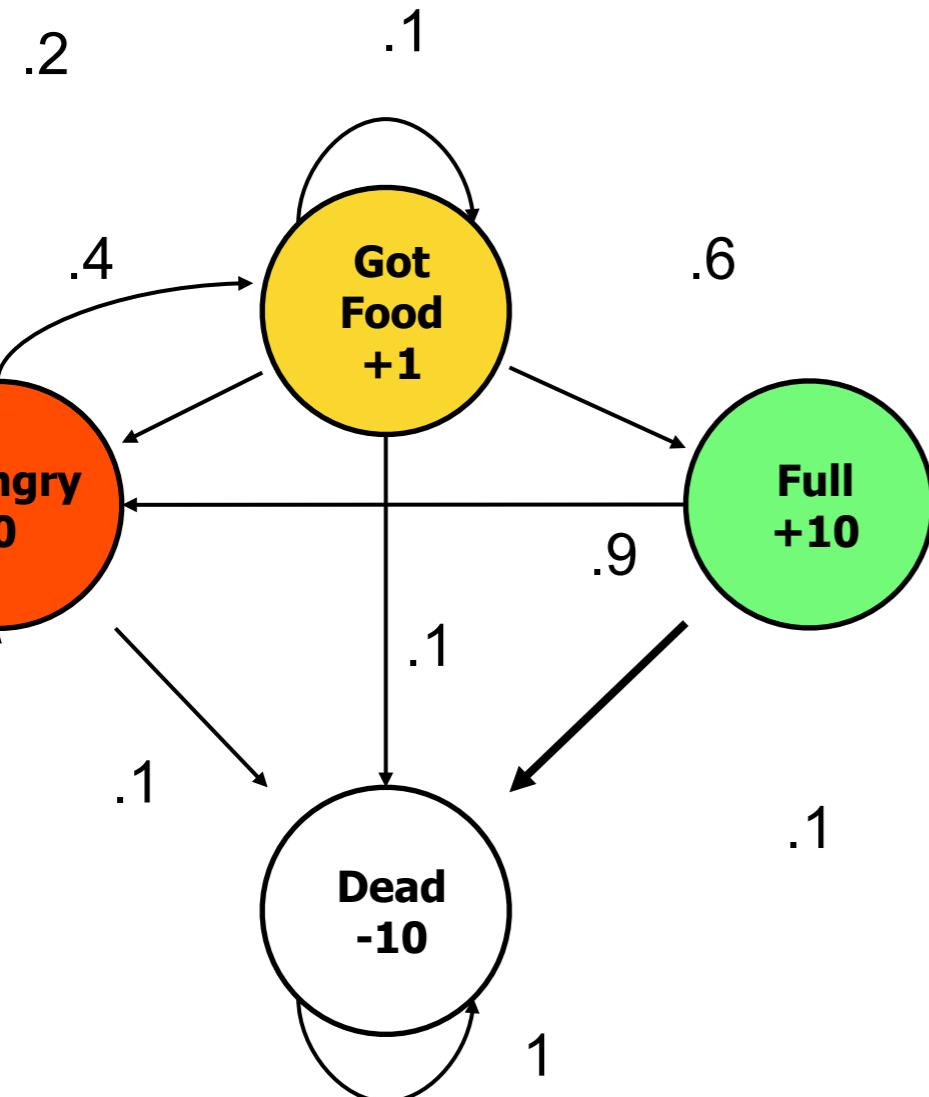
Value Iteration in Caveman's World



- Key idea: Bellman Recursion
- Relates value in current state to expected value of next state

$$\begin{aligned}V(s = H) &= r + \gamma(P_{HH}(R_H) + P_{HG}(R_G) + P_{HF}(R_F) + P_{HD}(R_D)) \\&= 0 + 0.9(0.5(0) + 0.4(1) + 0.1(10))\end{aligned}$$

Value Iteration in Caveman's World

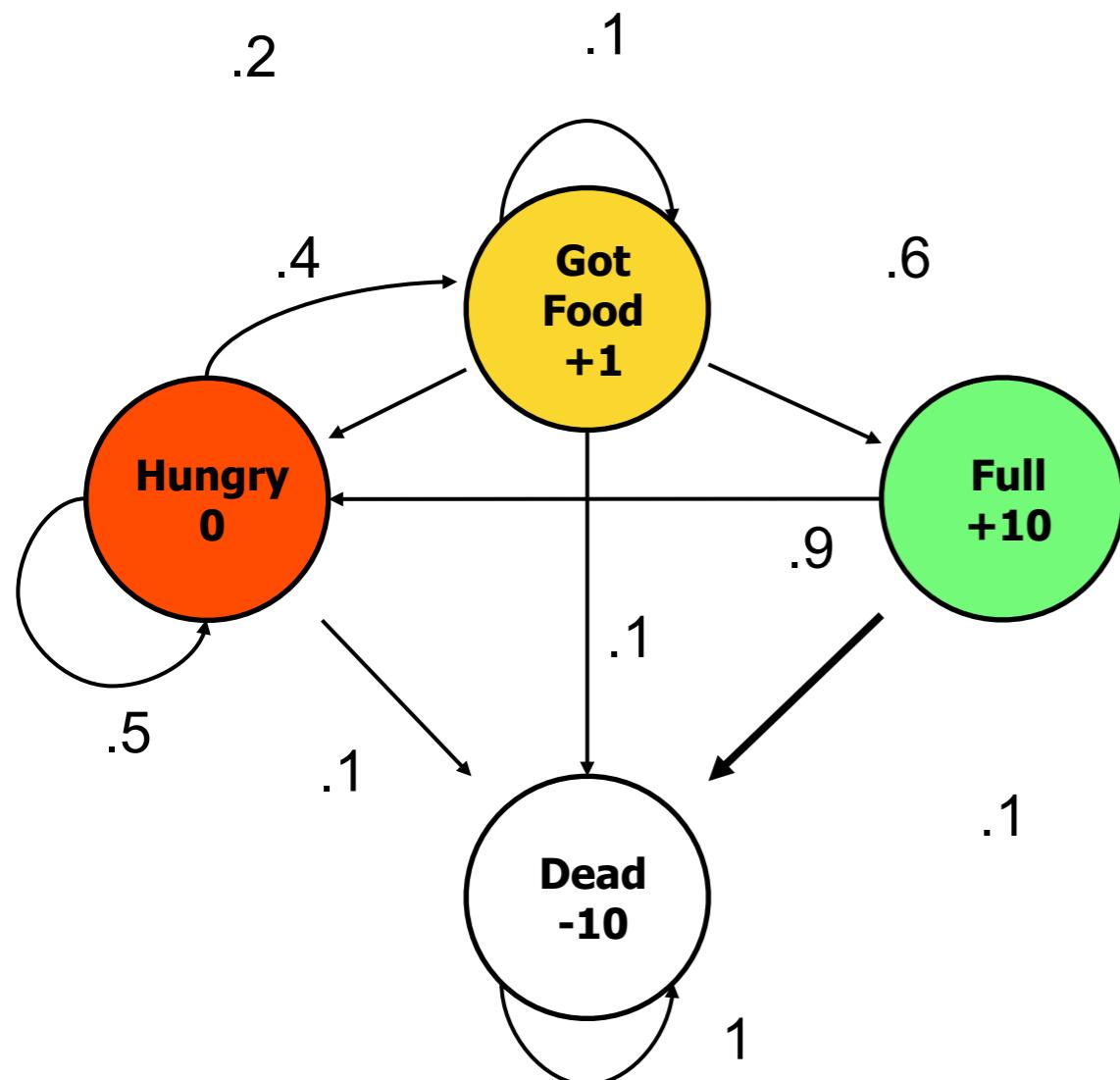


Value in k-steps

	H	G	F	D
1	0	1	10	-10
2	-0.54			

$$\begin{aligned}
 V(s = H) &= r + \gamma(P_{HH}(R_H) + P_{HG}(R_G) + P_{HF}(R_F) + P_{HD}(R_D)) \\
 &= 0 + 0.9(0.5(0) + 0.4(1) + 0.1(10))
 \end{aligned}$$

Value Iteration in Caveman's World



Value in k-steps

	H	G	F	D
1	0	1	10	-10
2	-.54	5.69	9.1	-19
3	.06	4.61	7.85	-27.1
4	-.75	3.23	7.61	-34.39
		...		
99	-39.08	-34.71	-30.66	-100.0
100	-39.09	-34.71	-30.66	-100.0

Value Iteration is Guaranteed to Converge

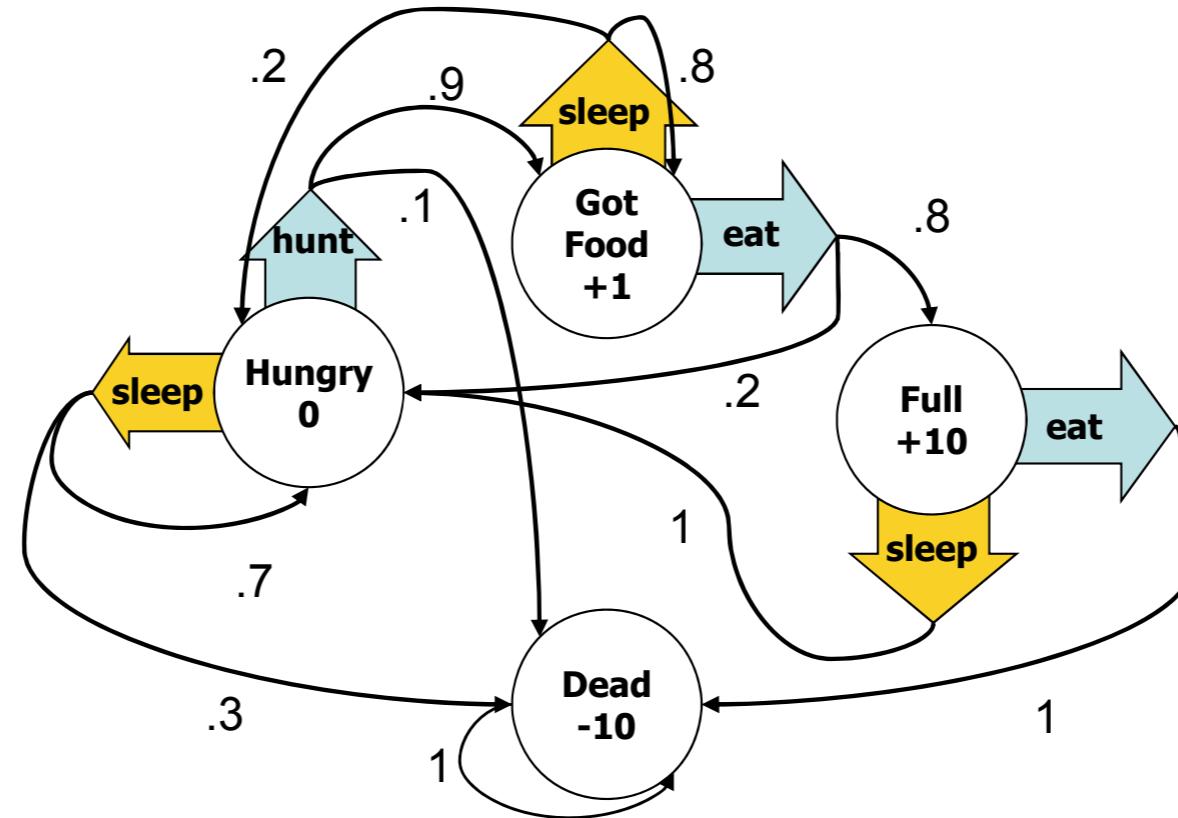
Summary

- Markov Processes represent **uncertainty in state transitions**
- It is possible to determine the **overall value of a state**
- What's next? Adding actions!

Actions: the value of free-will

- What'd we do so far?
 - Define values of states, and *transition probabilities* between them
 - To add actions, what do we need to look at?
 1. condition on actions: $P(s'|s) \rightarrow P(s'|s,a)$
 2. values of actions: $V(s) = \max_a Q(s,a)$
 - Turns out we need only (1), and (2) is RL

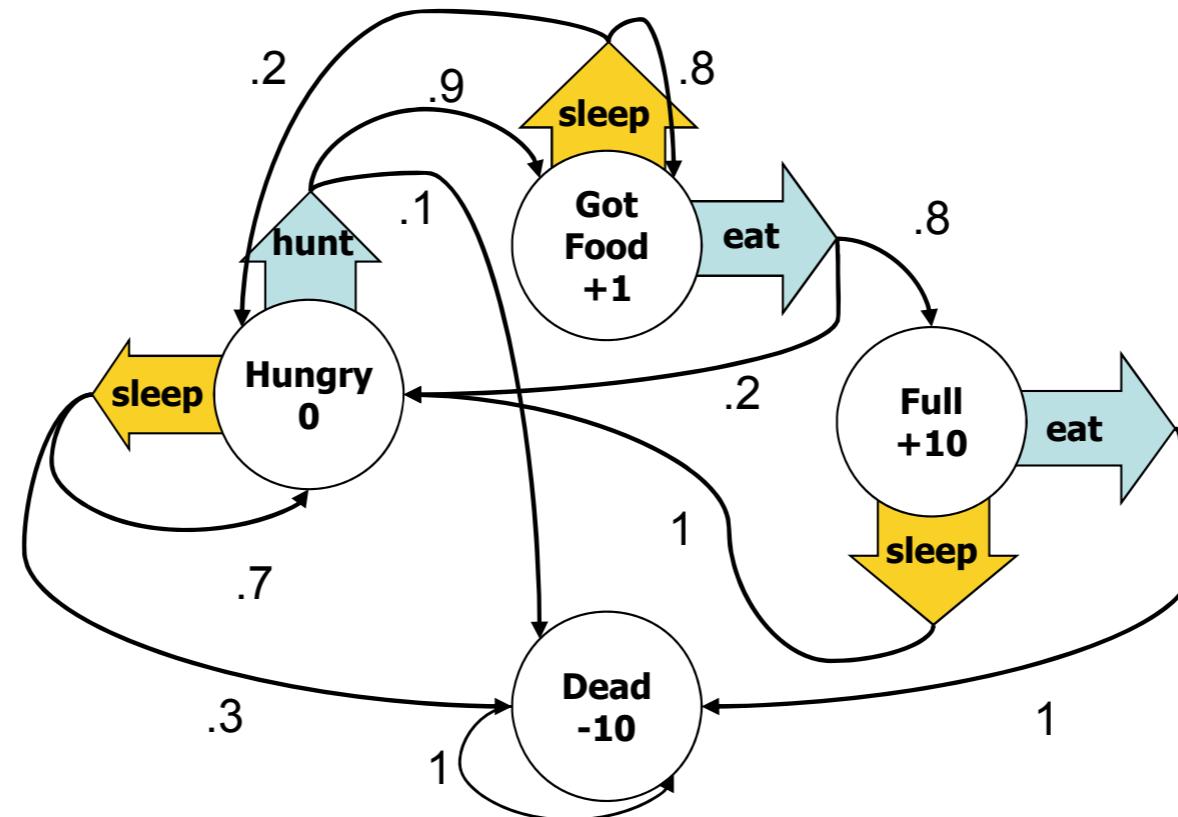
Actions: the value of free-will



- Adding actions back into an MDP:
- How? Make transitions conditional on action

$$T(s, a, s') = P(s'|s, a) = \begin{bmatrix} P_{11}^a & P_{12}^a & \dots & P_{1n}^a \\ P_{21}^a & P_{22}^a & \dots & P_{2n}^a \\ \vdots & \vdots & \ddots & \vdots \\ P_{n1}^a & P_{n2}^a & \dots & P_{nn}^a \end{bmatrix}$$

Actions: the value of free-will



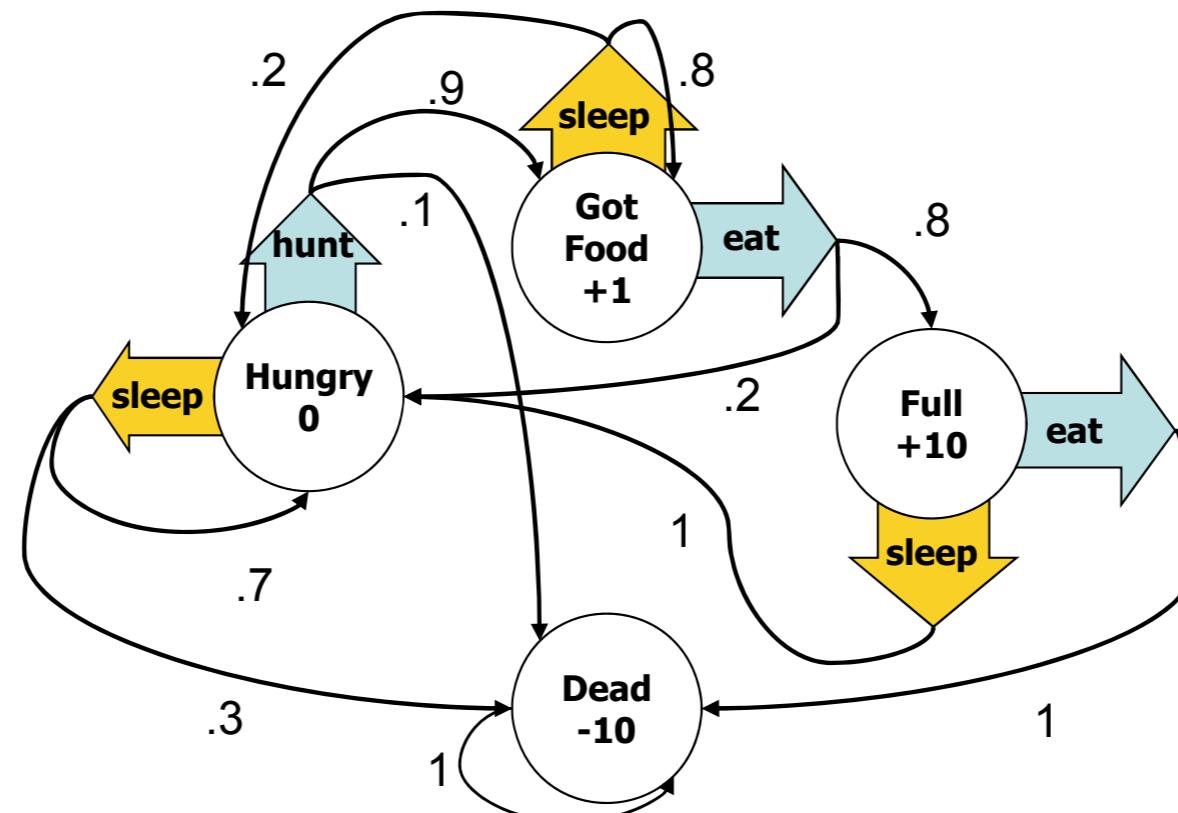
Value-Iteration needs one more thing:

$$V(s) \leftarrow \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V(s') \right]$$

*Bellman Equation

added this max over actions

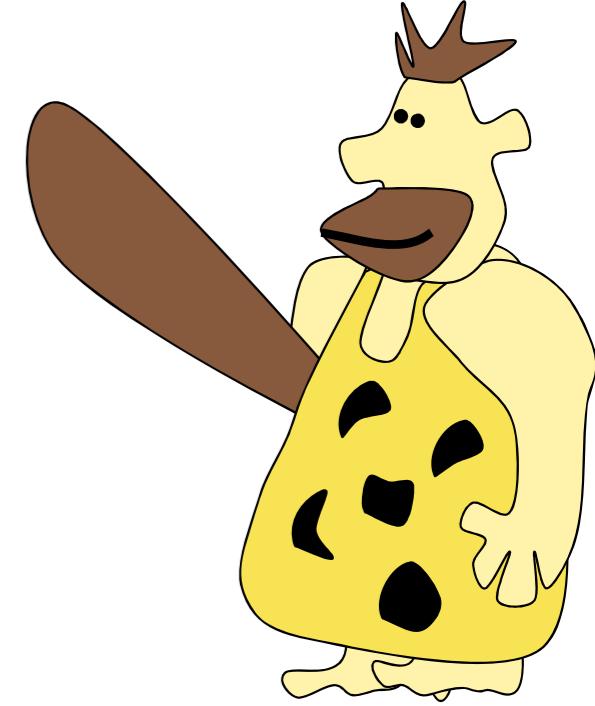
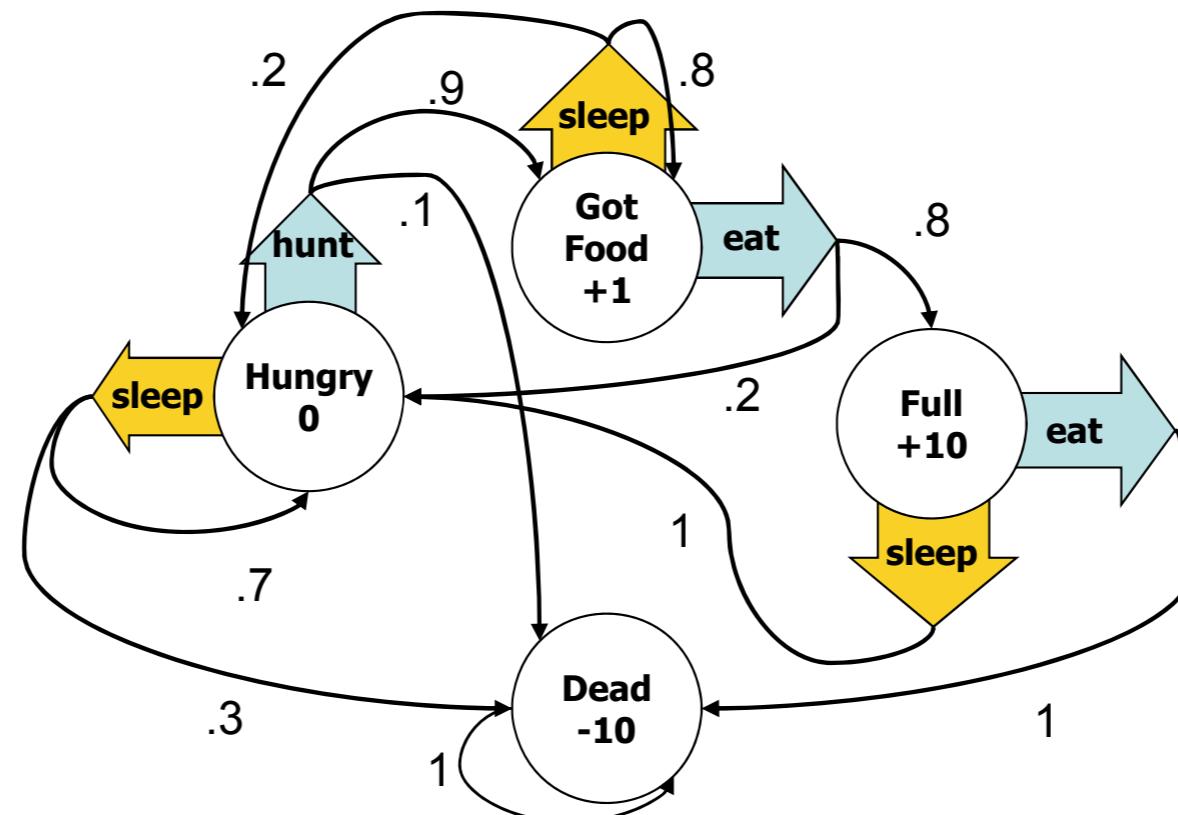
Actions: the value of free-will



“Free-Will” Values:

	H	G	F	D
1	0	1	10	-10
2	-.09	8.2		

Actions: the value of free-will



“Free-Will” Values:

	H	G	F	D
1	0	1	10	-10
2	-.09	8.2	10	-19
100	-7.16	2.27	3.56	-100

Value Iteration in Code

```
initialize  $V(s)$  arbitrarily
loop until policy good enough
    loop for  $s \in \mathcal{S}$ 
        loop for  $a \in \mathcal{A}$ 
            
$$Q(s, a) := R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s')V(s')$$

            
$$V(s) := \max_a Q(s, a)$$

        end loop
    end loop
```

What's this "Q" function?
→ Topic for later, but short answer is to allow action selection without lookahead

MDP Planning: Core concepts

- Things to really understand about MDPs:
 - ➡ what a value function is
 - ➡ why we can converge to V^* with these simple algorithms
 - ➡ why V^* is overkill sometimes
 - ➡ why model is so important, and what to do without it
 - ➡ why these algorithms can be (horribly) inefficient

Value Iteration: Big Questions

- Convergence?
- Efficiency?
- Assumptions?

Value Iteration Convergence

- Proof Sketch:

- I. Defined in terms of max-norm between any two value functions (in particular V_i and V^*)
2. Take advantage of basic property of max:

$$|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|$$

3. Apply Bellman operator and rearrange

$$\begin{aligned} |B(V_i) - B(V_j)|(s) &= \left| \left(R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) V_i(s') \right) - \left(R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) V_j(s') \right) \right| \\ &= \gamma (\max_a E_{V_i}[s'] - \max_a E_{V_j}[s']) \\ &\leq \gamma \max_a (E_{V_i}[s'] - E_{V_j}[s']) \\ &\leq \gamma \max_a (V_i(s') - V_j(s')) \end{aligned}$$

tl;dr: max-norm (max difference w.r.t. V^*) strictly contracts with each application of Bellman (with factor gamma)

But how important is convergence?

- Why does value matter again? To pick actions
 - IE, we're interested in $\pi(s)$, not $V(s)$
- Can we optimize the policy directly?
 - Yes! This is “policy iteration”
 - We'll use the policy form of Bellman:

$$V_{t+1}^{\pi}(s) \leftarrow R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V_t^{\pi}(s')$$

Policy Iteration

- Alternative approach:
 - Step 1: Policy evaluation: calculate value for some fixed policy (not optimal utilities!) until convergence
 - Step 2: Policy improvement: update policy using onestep look-ahead with resulting converged (but not optimal!) utilities as future values
 - Repeat steps until policy converges (it does)
- Facts about policy iteration:
 - It's still optimal!
 - Can converge faster under some conditions. Why??

Implementing Policy Iteration

- Simple change:
 - I. Evaluate policy somehow
 - ➡ option 1: solve as linear system
 - ➡ option 2: use Bellman for a while

$$V_0^\pi(s) \leftarrow 0$$

$$V_{t+1}^\pi(s) \leftarrow R(s, \pi_t(s)) + \gamma \sum_{s'} P(s'|s, \pi_t(s)) V_t^\pi(s')$$

2. Improve policy using 1-step lookahead

$$\pi_{k+1}^*(s) = \arg \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^{\pi_k}(s') \right]$$

Policy Iteration Convergence

Policy iteration convergence proof sketch:

- (1) In every step the policy improves. Means that a given policy can be encountered at most once. This means that after we have iterated as many times as there are different policies (i.e., $|A|^{|S|}$), we must be done and hence have converged.
- (2) By definition at convergence we have that $\pi_{k+1}(s) = \pi_k(s) \quad \forall s \in S$. This implies that $V^{\pi_k} = \max_a [R(s, a) + \gamma \sum_{s'} P(s'|s, a)V^{\pi_k}(s')]$ for all states. This satisfies the Bellman equation, which means V^{π_k} is equal to the optimal value function V^* .

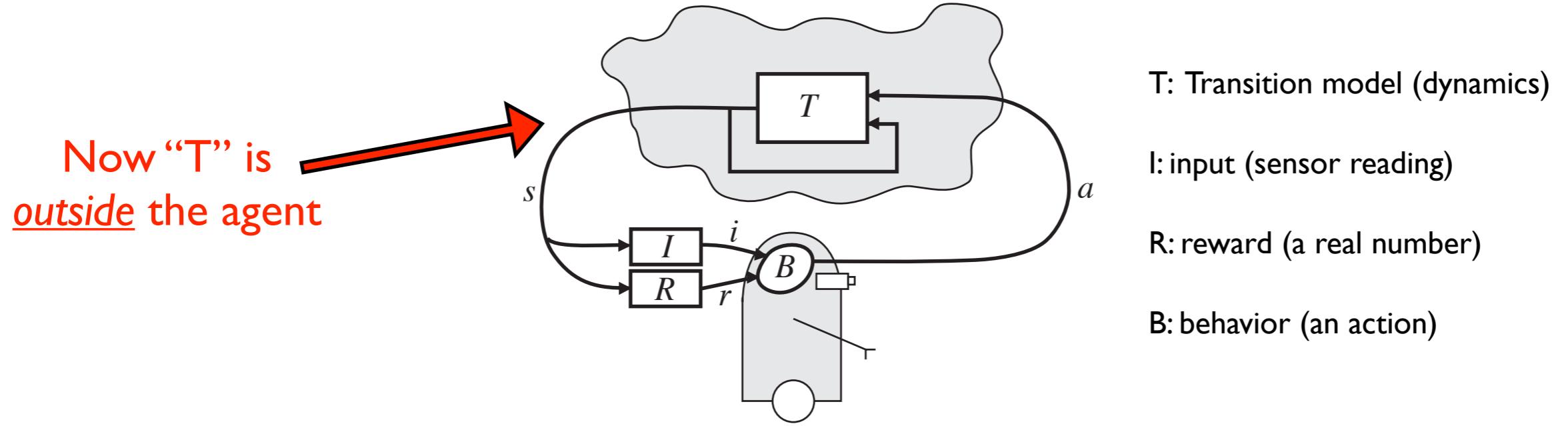
Comparison to Value Iteration

- What's the real difference vs. VI?
 - Just puts more effort into policy evaluations in between policy updates
 - Why might this be helpful??
 - ➡ Early convergence criterion (policy stops changing)
 - ➡ When we have lots of actions, so update is expensive

Reinforcement Learning

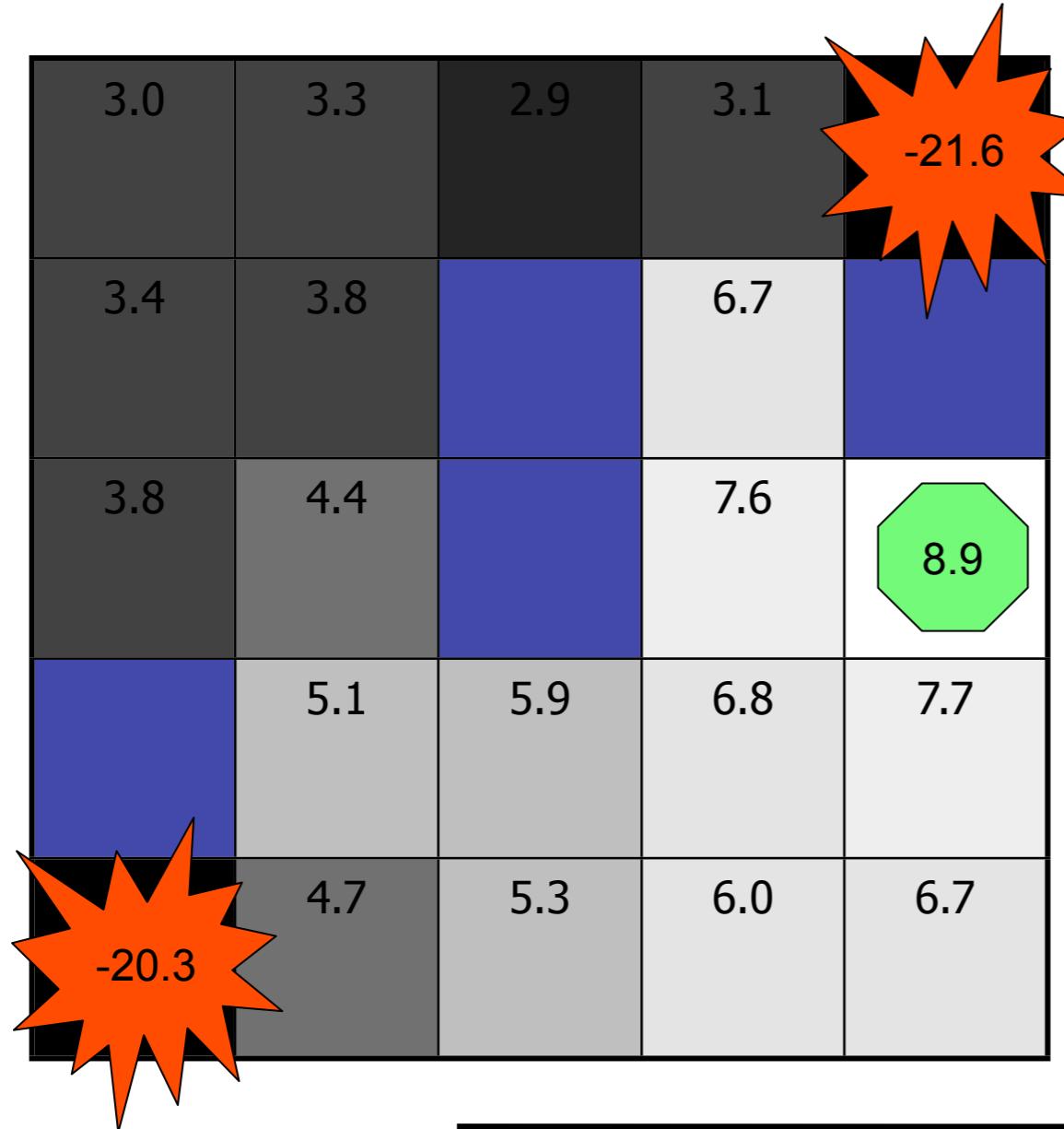
- Notice: all previous methods required the model
 - What if we don't have it? Can we learn from pure exploration??
- Yes! This is “reinforcement learning”
- Today we'll derive Q-learning, simplest model-free RL algorithm

Life of an RL Agent



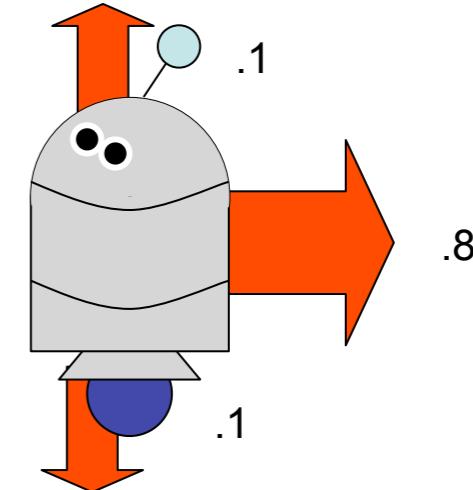
- Agent lives in loop:
 1. receive observation (eg camera image)
 2. select action
 3. receive reward

Review of MDPs

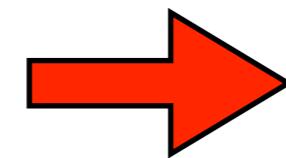


Optimal Value Function

Solved with
value iteration



How do we use
 $V(s)$ for planning?



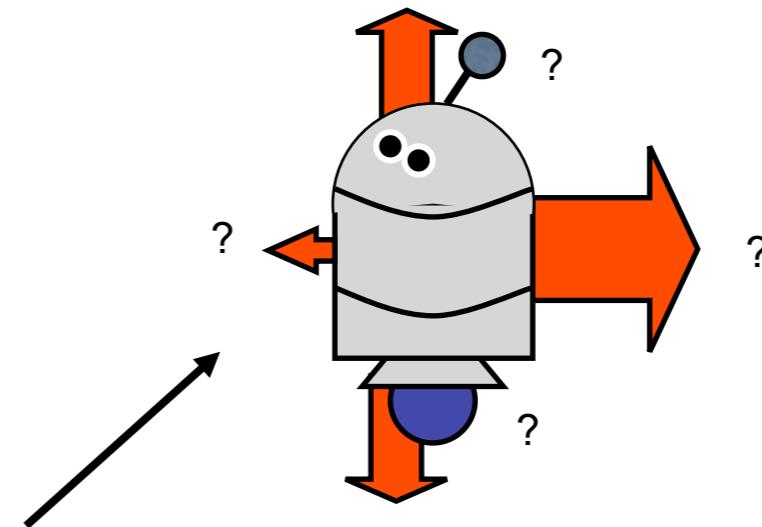
$$\pi^*(s) = \arg \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a)V(s') \right]$$

(1-step look-ahead)

We assumed a model for $P(s'|s,a)$

**What do we do if such a model
does not exist?**

$P(s'|s,a)$ unknown!

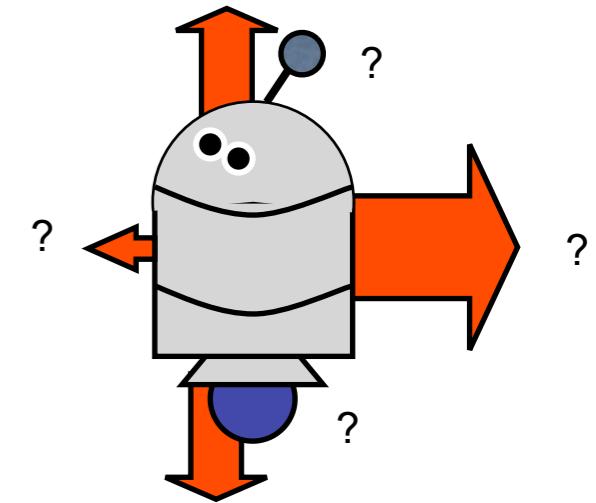


Model (without walls)

We assumed a model for $P(j|i,a)$

What do we do if such a model does not exist?

- Learn one (e.g. Bayesian RL)
- “Model-Free” RL (e.g. Q-Learning)



Want: $\pi^*(s)$ – Optimal policy in the state

Can't use: $V^*(s)$ – Value of a state

Learn instead: $Q^*(s_i, a)$ - Value of taking an action in a state

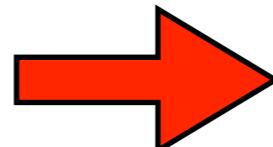
Q-function Definition

Key Relationship:

$$V(s) = \max_a Q(s, a)$$

$Q(s, a)$ = Value of Taking action **a** in state **S**

How do we use
 $Q(s, a)$ for planning?



$$\pi^*(s) = \arg \max_a Q(s, a)$$

Q-function Definition

Definition of Q function:

$$\begin{aligned} Q(s, a) &= R(s, a) + \gamma \max_{a'} \mathbb{E} [Q(s', a')] \\ &= R(s, a) + \gamma \max_{a'} \sum_{s'} P(s'|s, a) Q(s', a') \end{aligned}$$

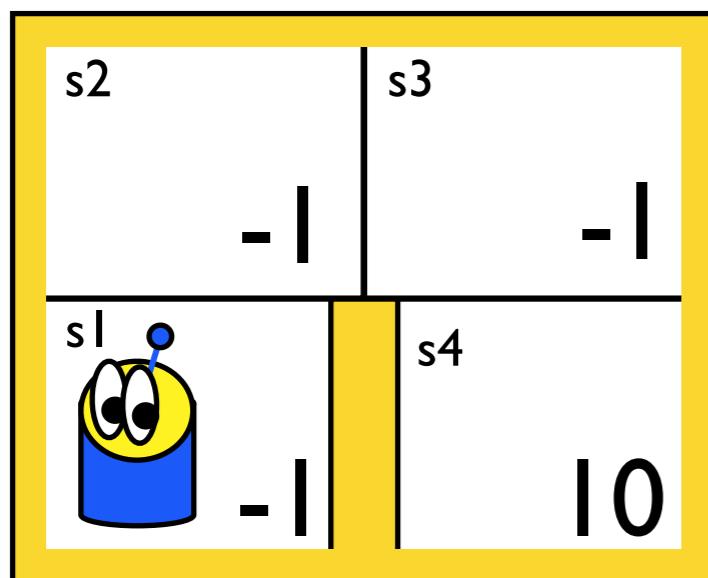
How to remove dependency on model?

From Q-function to Q-Learning

→ Key question: How to remove dependency on model?

$$\begin{aligned} Q(s, a) &= R(s, a) + \gamma \max_{a'} \sum_{s'} P(s'|s, a) Q(s', a') && \text{by definition} \\ &\approx R(s, a) + \gamma \max_{a'} Q(s', a'), \quad s' \sim P(s'|s, a) && \text{by sample approximation} \\ &\approx (1 - \alpha)Q(s, a) + \alpha \left(R(s, a) + \gamma \max_{a'} Q(s', a') \right) && \text{smoothing} \\ &\approx Q(s, a) - \alpha Q(s, a) + \alpha R(s, a) + \alpha \gamma \max_{a'} Q(s', a') \\ &\approx Q(s, a) + \alpha \left(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) && \text{canonical form} \\ &\approx Q(s, a) + \alpha(\delta_{TD}) && \text{TD error} \end{aligned}$$

Q-Learning Example

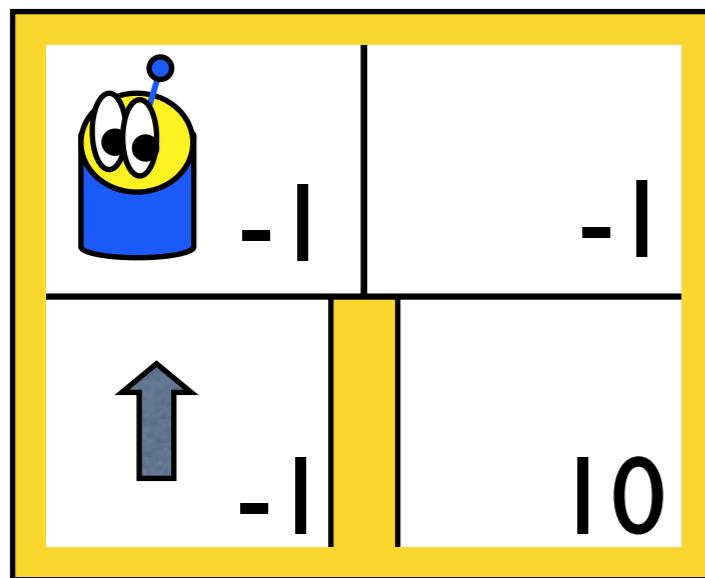


$$\alpha = .7$$

	↑	↓	←	→
S₁	0	0	0	0
S₂	0	0	0	0
S₃	0	0	0	0
S₄	0	0	0	0

Q-Table

Q-Learning Example

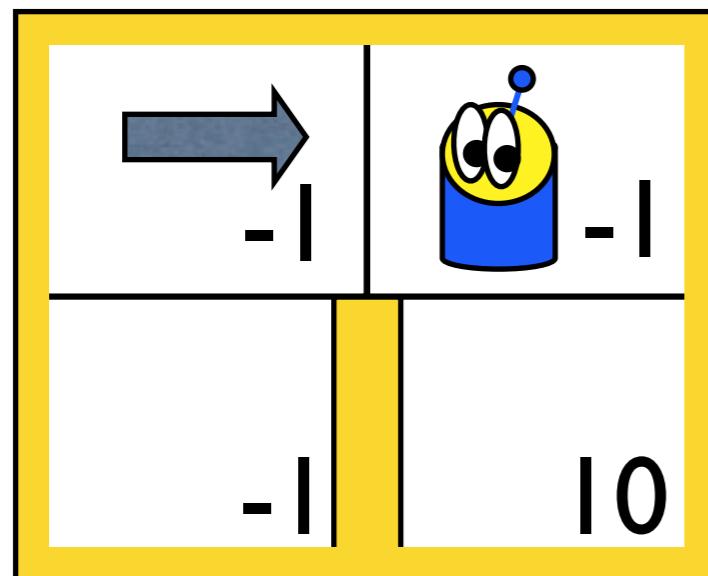


$$Q^{\text{est}}(S_1, \uparrow) = .7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	\uparrow	\downarrow	\leftarrow	\rightarrow
S_1	- .7	0	0	0
S_2	0	0	0	0
S_3	0	0	0	0
S_4	0	0	0	0

Q-Table

Q-Learning Example

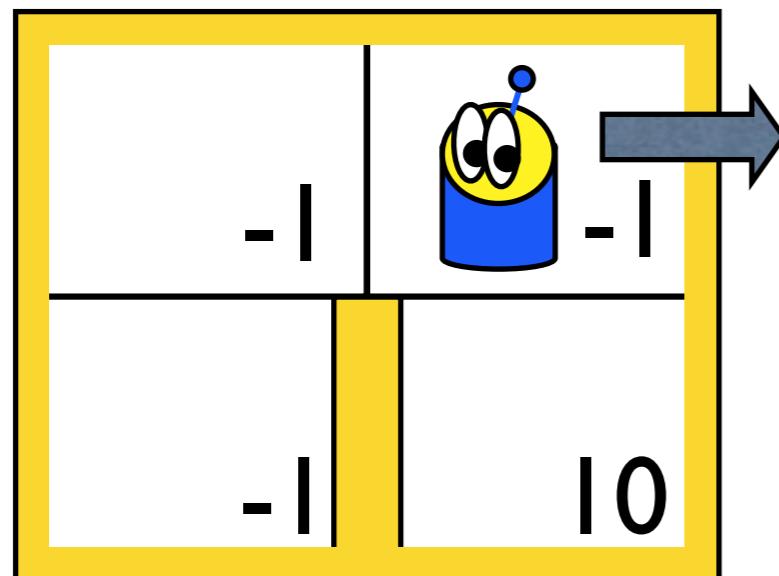


$$Q^{\text{est}}(S_2, \rightarrow) = .7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	↑	↓	←	→
S₁	- .7	0	0	0
S₂	0	0	0	- .7
S₃	0	0	0	0
S₄	0	0	0	0

Q-Table

Q-Learning Example

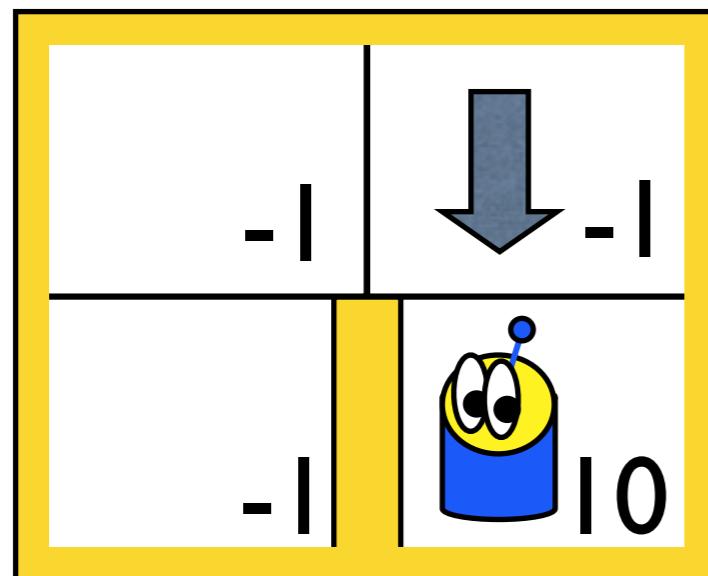


$$Q^{\text{est}}(S_3, \rightarrow) = .7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	↑	↓	←	→
S₁	- .7	0	0	0
S₂	0	0	0	- .7
S₃	0	0	0	- .7
S₄	0	0	0	0

Q-Table

Q-Learning Example

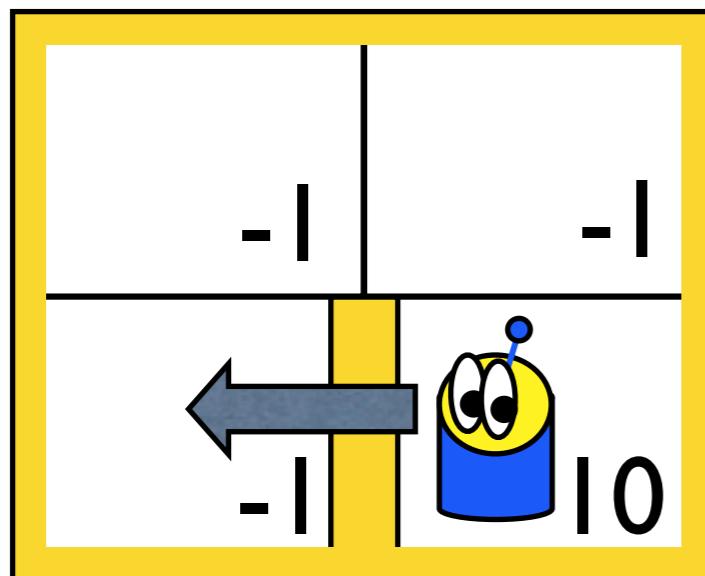


$$Q^{\text{est}}(S_3, \downarrow) = .7(-1 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	↑	↓	←	→
S₁	- .7	0	0	0
S₂	0	0	0	- .7
S₃	0	- .7	0	- .7
S₄	0	0	0	0

Q-Table

Q-Learning Example

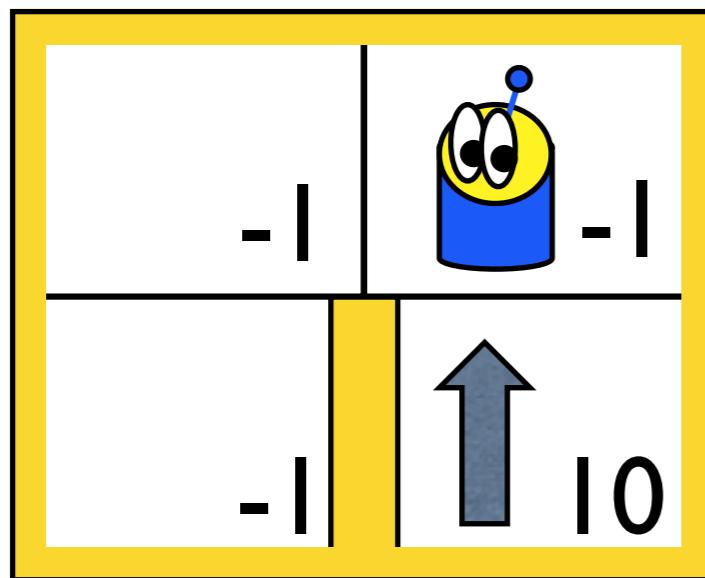


$$Q^{\text{est}}(S_4, \leftarrow) = .7(10 + .9 \max(0, 0, 0, 0)) + .3 \times 0$$

	↑	↓	←	→
S₁	- .7	0	0	0
S₂	0	0	0	- .7
S₃	0	- .7	0	- .7
S₄	0	0	7	0

Q-Table

Q-Learning Example

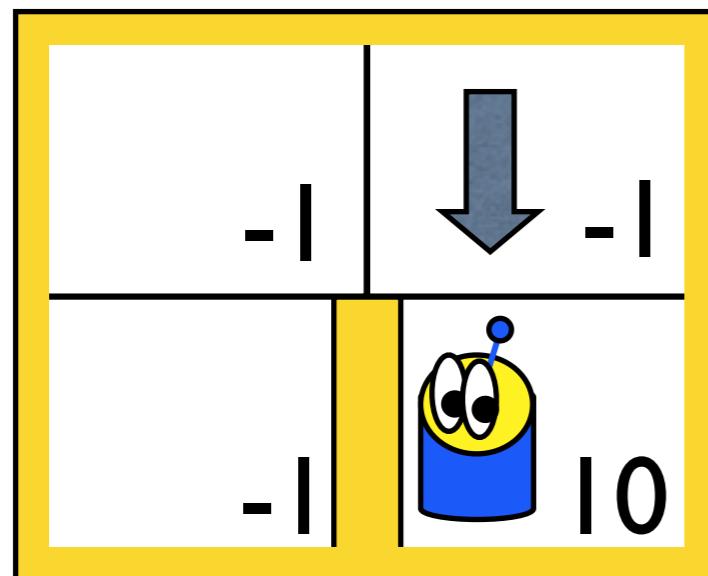


$$Q^{\text{est}}(S_4, \uparrow) = .7(10 + .9 \max(0, -.7, 0, -.7)) + .3 \times 0$$

	\uparrow	\downarrow	\leftarrow	\rightarrow
S_1	-.7	0	0	0
S_2	0	0	0	-.7
S_3	0	-.7	0	-.7
S_4	7	0	7	0

Q-Table

Q-Learning Example



$$Q^{\text{est}}(S_3, \downarrow) = .7(-1 + .9 \max(7, 0, 7, 0)) + .3 \times -.7$$

	↑	↓	←	→
S₁	-.7	0	0	0
S₂	0	0	0	-.7
S₃	0	3.5	0	-.7
S₄	7	0	7	0

Q-Table

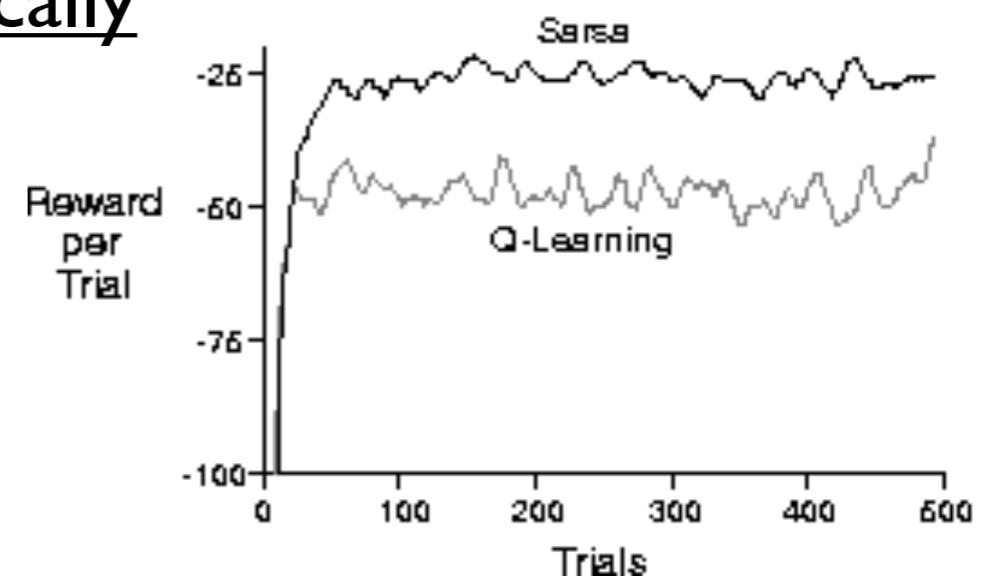
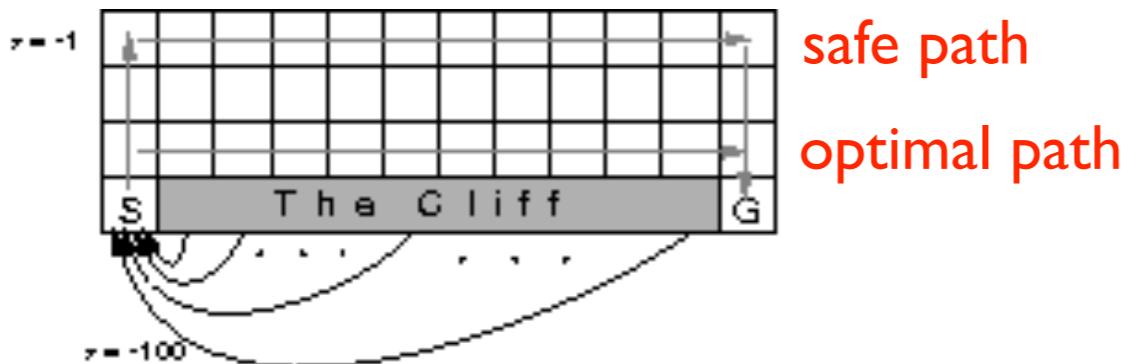
On-Policy Learning: SARSA

- Key idea: perform backups on action actually selected, rather than estimate of optimal action

$$Q(s, a) \leftarrow Q(s, a) + \alpha (R(s, a) + \gamma Q(s', a') - Q(s, a))$$

↑ ↑ ↑ + ↑ ↑
state + action + reward + (next) state + (next) action = **SARSA**

- Otherwise same as Q-learning, but “on-policy”
- Less greedy, so addresses problem of locally high-reward/risk states (e.g. cliff task)



TD and eligibility traces

- Problem: Q-values spread slowly
- Solution: Propagate over history
- Mechanism: exponential decay w.p. λ

TD error for last action

$$\delta_{td} = R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

TD error at time T-t

$$e(s, a) = \gamma^t \lambda^t \delta_{td}$$

SARSA(λ)

Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0$, for all s, a

Repeat (for each episode):

 Initialize s, a

 Repeat (for each step of episode):

 Take action a , observe r, s'

 Choose a' from s' using policy derived from Q
(e.g., ϵ -greedy)

$$\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$$

$$e(s, a) \leftarrow e(s, a) + 1$$

 For all s, a :

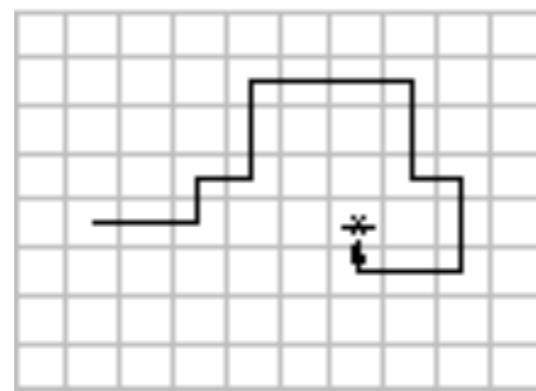
$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$$

$$e(s, a) \leftarrow \gamma \lambda e(s, a)$$

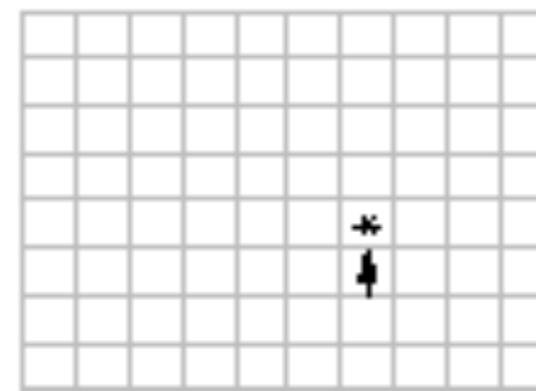
$$s \leftarrow s'; a \leftarrow a'$$

until s is terminal

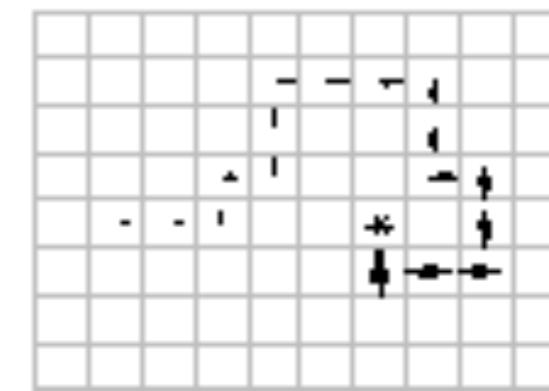
This is the TD-error



Path Taken



action values increased
by 1-step SARSA



action values increased by
SARSA(λ), $\lambda=0.9$

Take-home

- Use Q-learning/SARSA when:
 - state space is tiny
 - interested in full policy
 - don't have access to model
- Use eligibility traces when:
 - always.

Monte-Carlo Reinforcement Learning

- Recall:
 - V.I., P.I., Q-Learning, & SARSA are all direct implementations of bellman recursion, via *dynamic programming*
- MCRL is direct implementation of reward expectation, via sampling

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^T \gamma^t r_t \right]$$

- Returns are simply averaged together
- Variance of the error decreases as $1/n$

Monte-Carlo Reinforcement Learning

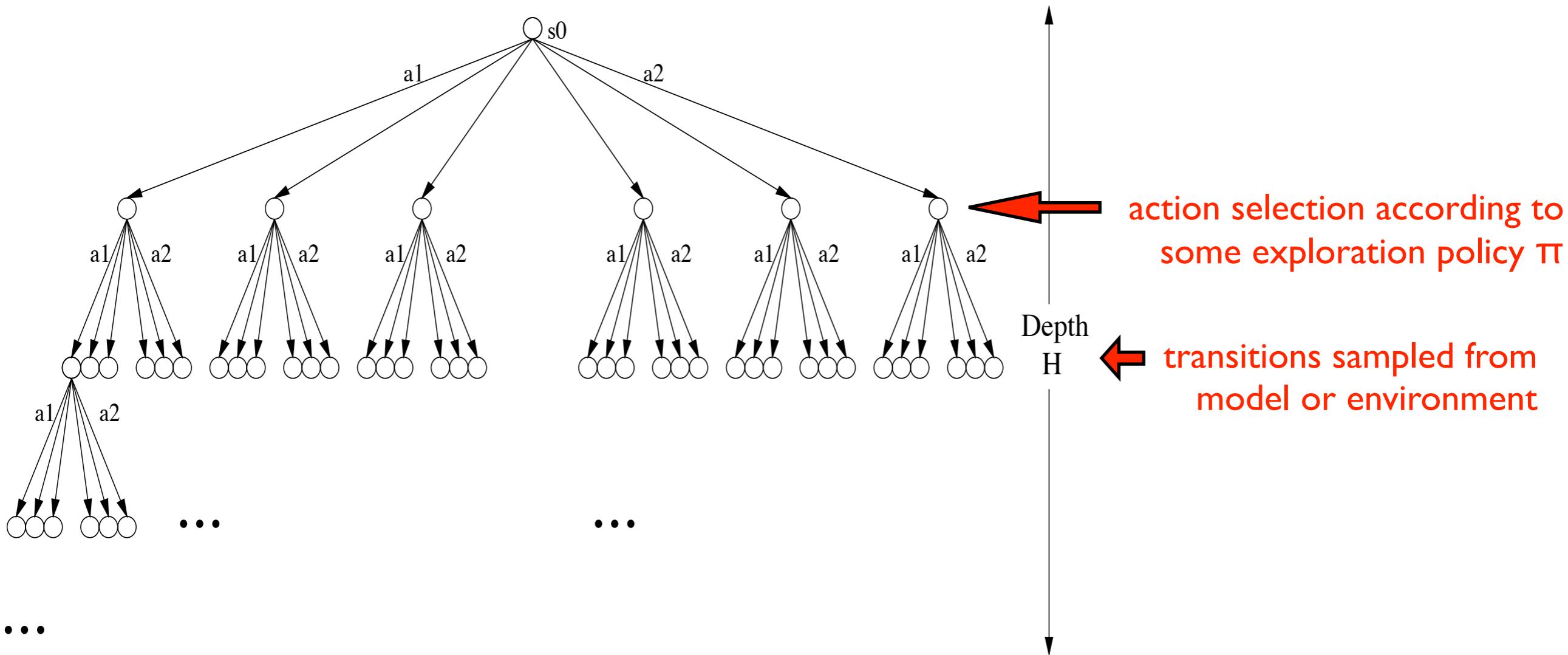
Unpacking the bellman recursion:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right] \\ &= \mathbb{E}_\pi \left[r_0 + \sum_{t=0}^{\infty} \gamma^{t+1} r_{t+1} | s_0 = s \right] \\ &= \mathbb{E}_\pi \left[r_0 + \gamma r_1 + \sum_{t=0}^{\infty} \gamma^{t+2} r_{t+2} | s_0 = s \right] \\ &= R(s) + \gamma \sum_a P_\pi(a|s) \sum_{s'} P(s'|s, a) \left[R(s') + \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^{t+1} r_{t+1} | s_0 = s' \right] \right] \\ &= R(s) + \gamma \sum_a P_\pi(a|s) \sum_{s'} P(s'|s, a) V(s') \quad \text{← back to bellman} \end{aligned}$$

The point: you can approximate bellman using finite sums

Monte-Carlo Reinforcement Learning

How to visualize:

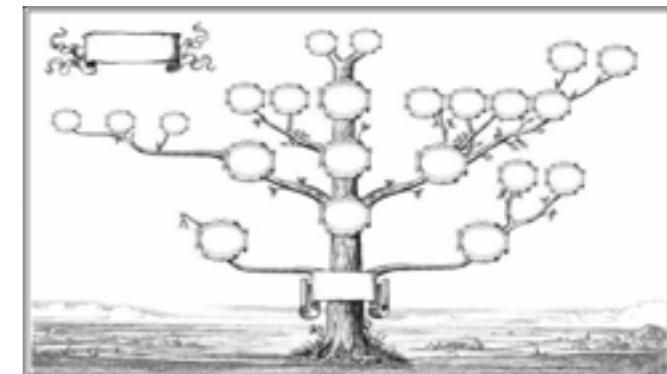


Monte-Carlo Reinforcement Learning

- Key properties:
 - runtime independent of $|S|$ (!)
 - can learn from actual and simulated experience
 - can target parts of the state space we care about (!)
- Problems:
 - Slow
 - When to stop?
 - Variance falls as $1/n$, can we do better?

Sparse-Sampling (MCTS)

- ONLINE MCRL algorithm with provable loss bounds
 - Kearns, Mansour, Ng (ML 2002)
- Key idea: rewards in future matter less than rewards now
- Outputs:
 - ε -optimal policy



$$|V^{\mathcal{A}}(s) - V^*(s)| \leq \varepsilon$$

Sparse-Sampling (MCTS)

Running Time: $O((kC)^H)$

- Hairy Math:

$$H = \lceil \log_\gamma(\lambda/V_{\max}) \rceil$$

Planning horizon

$$C = \frac{V_{\max}^2}{\lambda^2} \left(2H \log \frac{kHV_{\max}^2}{\lambda^2} + \log \frac{R_{\max}}{\lambda} \right)$$

Number of rollouts

$$\lambda = (\epsilon(1 - \gamma)^2)/4, V_{\max} = R_{\max}/(1 - \gamma)$$

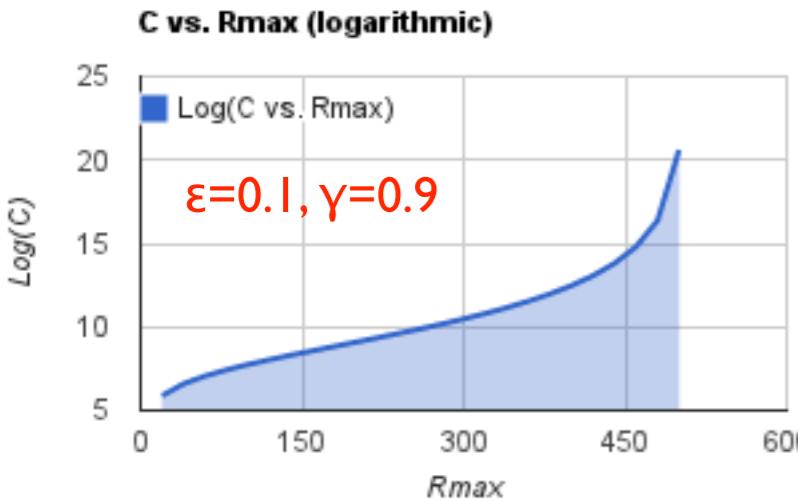
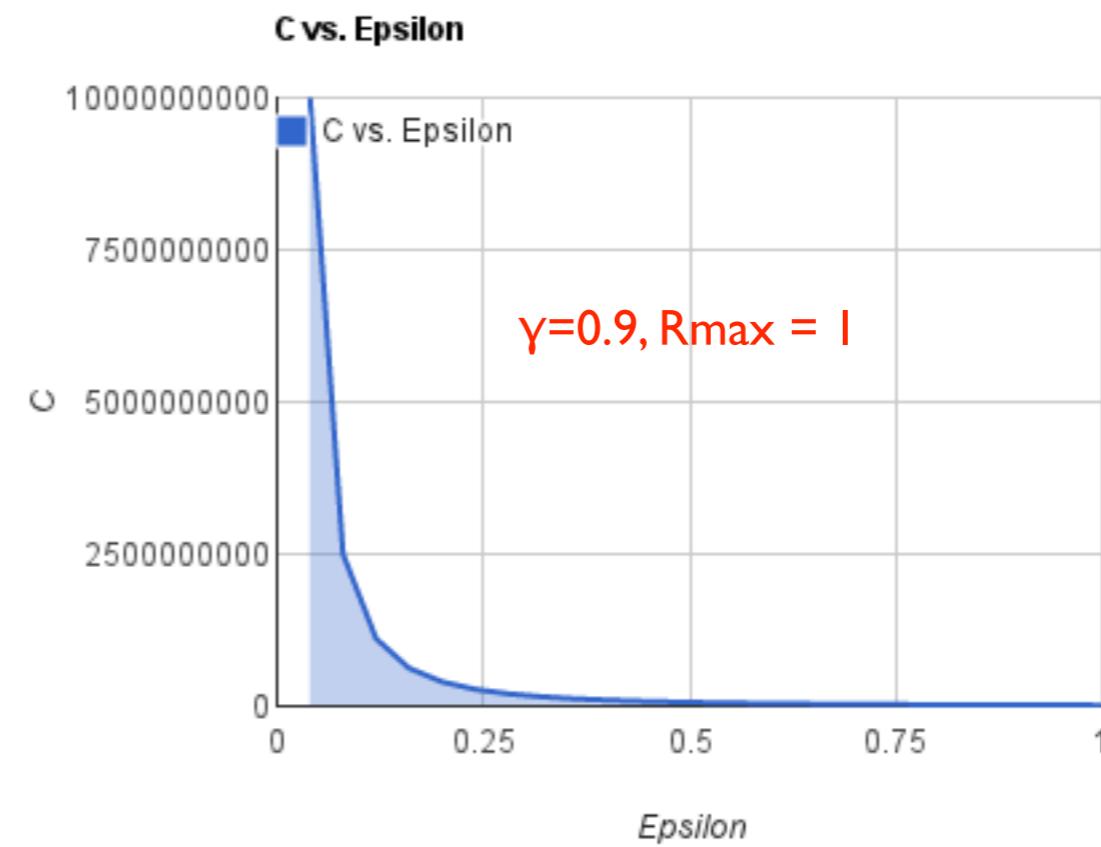
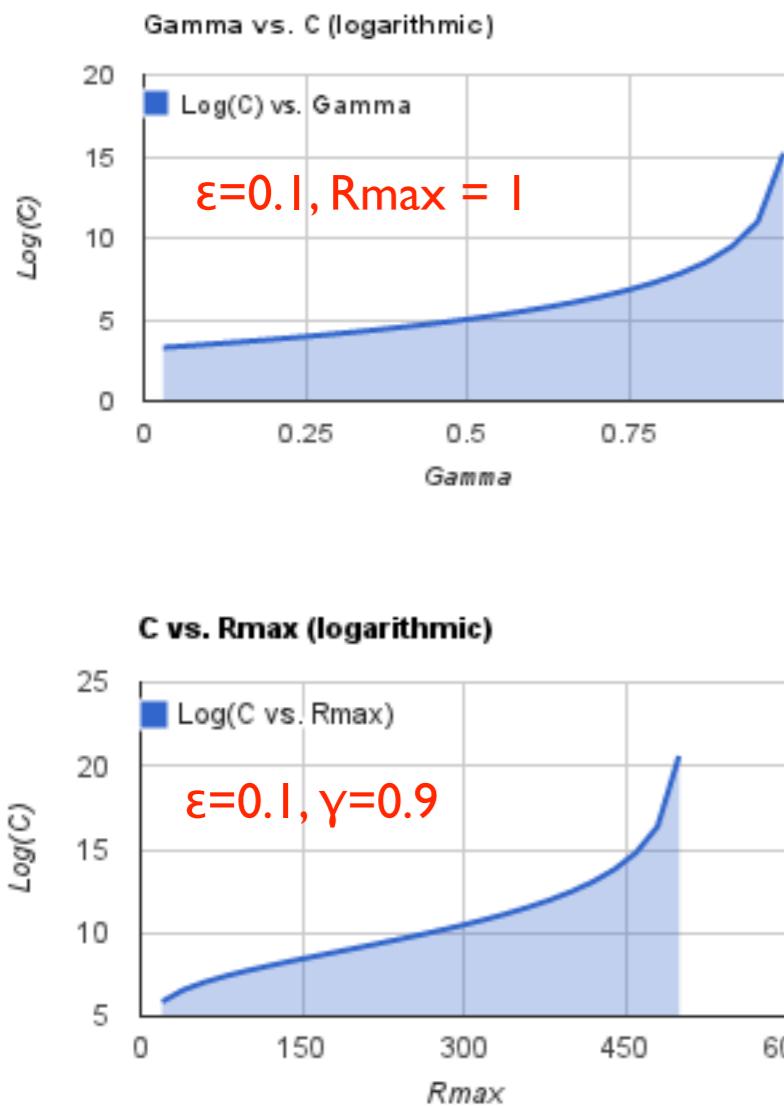
Useful Constants

- Running time depends only on R_{\max} , ϵ , and γ !

→ The point: can do MCRL with provable guarantees. But how useful??

Sparse-Sampling (MCTS)

- Problem: C can be HUGE



Take-home

- Use MCRL/MCTS when:
 - state space is huge
 - interested in subset of S (online planning)
 - planning horizon is small
 - can efficiently sample from model
- Related work:
 - UCT (Kocsis et al 2006) ← Reigning GO champion!
 - FSSS (Walsh et al 2010)