

Artificial Intelligence

CE-417, Group 1

Computer Eng. Department

Sharif University of Technology

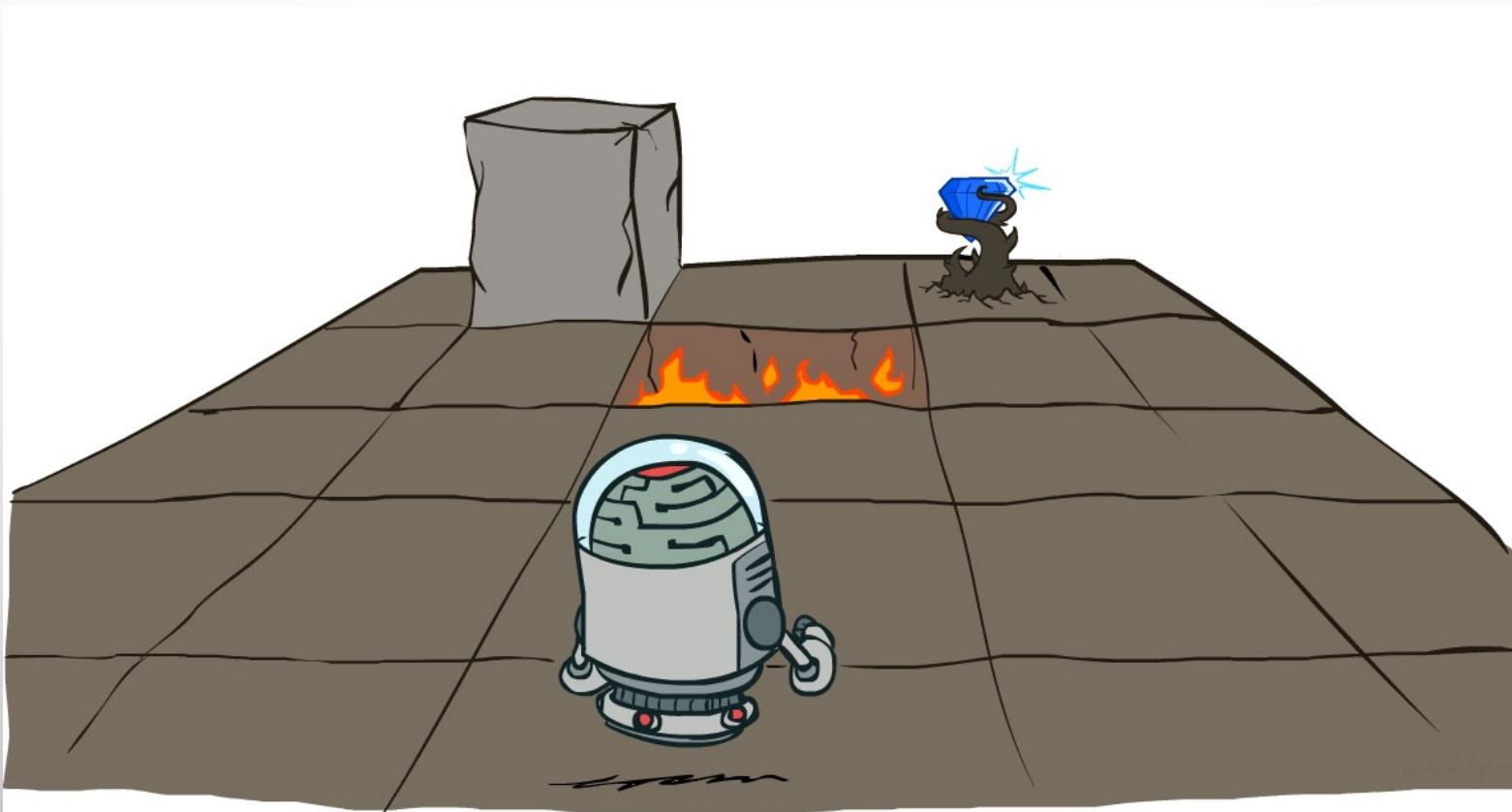
Spring 2023

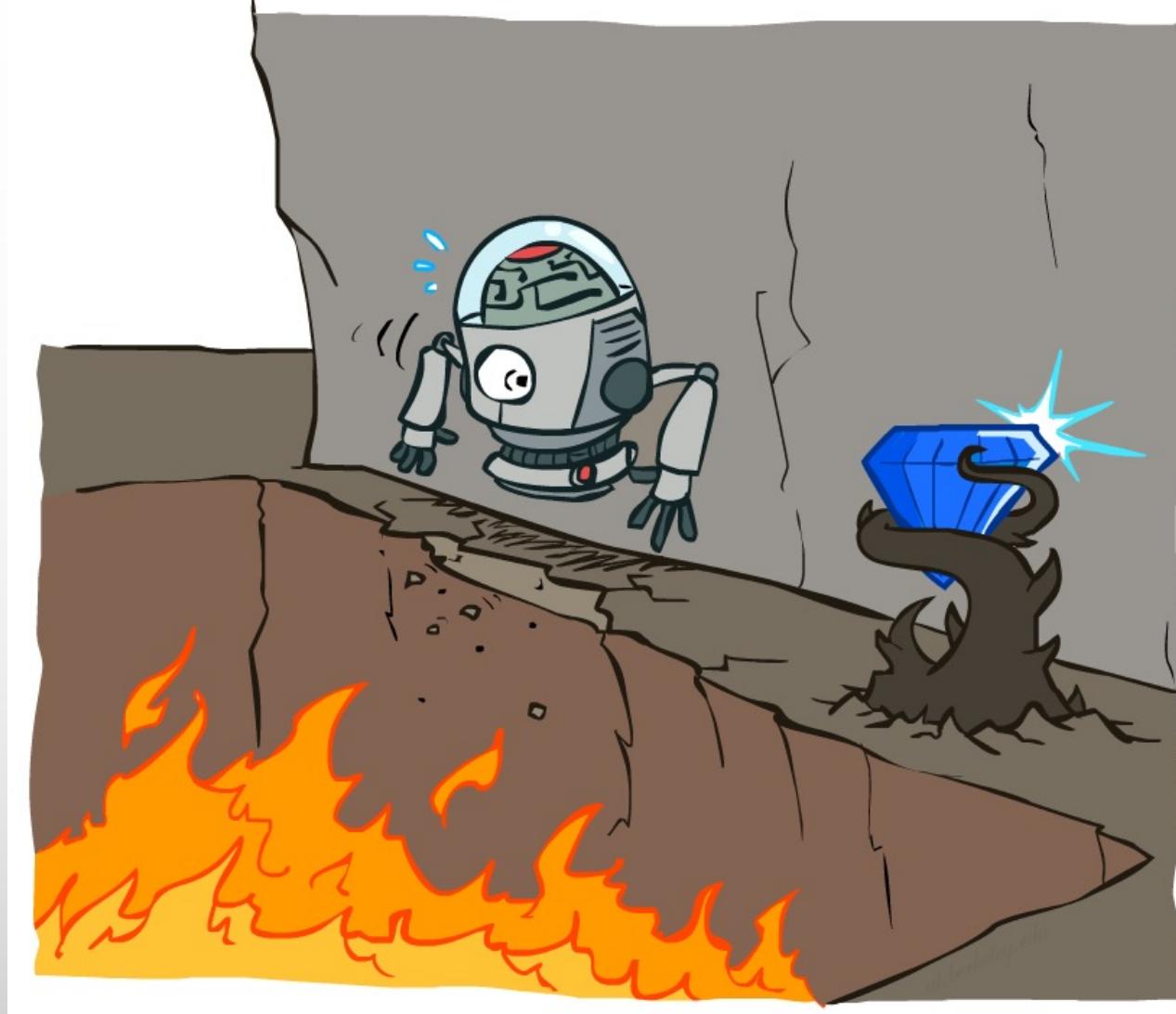
By Mohammad Hossein Rohban, Ph.D.

Courtesy: Most slides are adopted from CSE-573 (Washington U.), original
slides for the textbook, and CS-188 (UC. Berkeley).

Markov Decision Processes

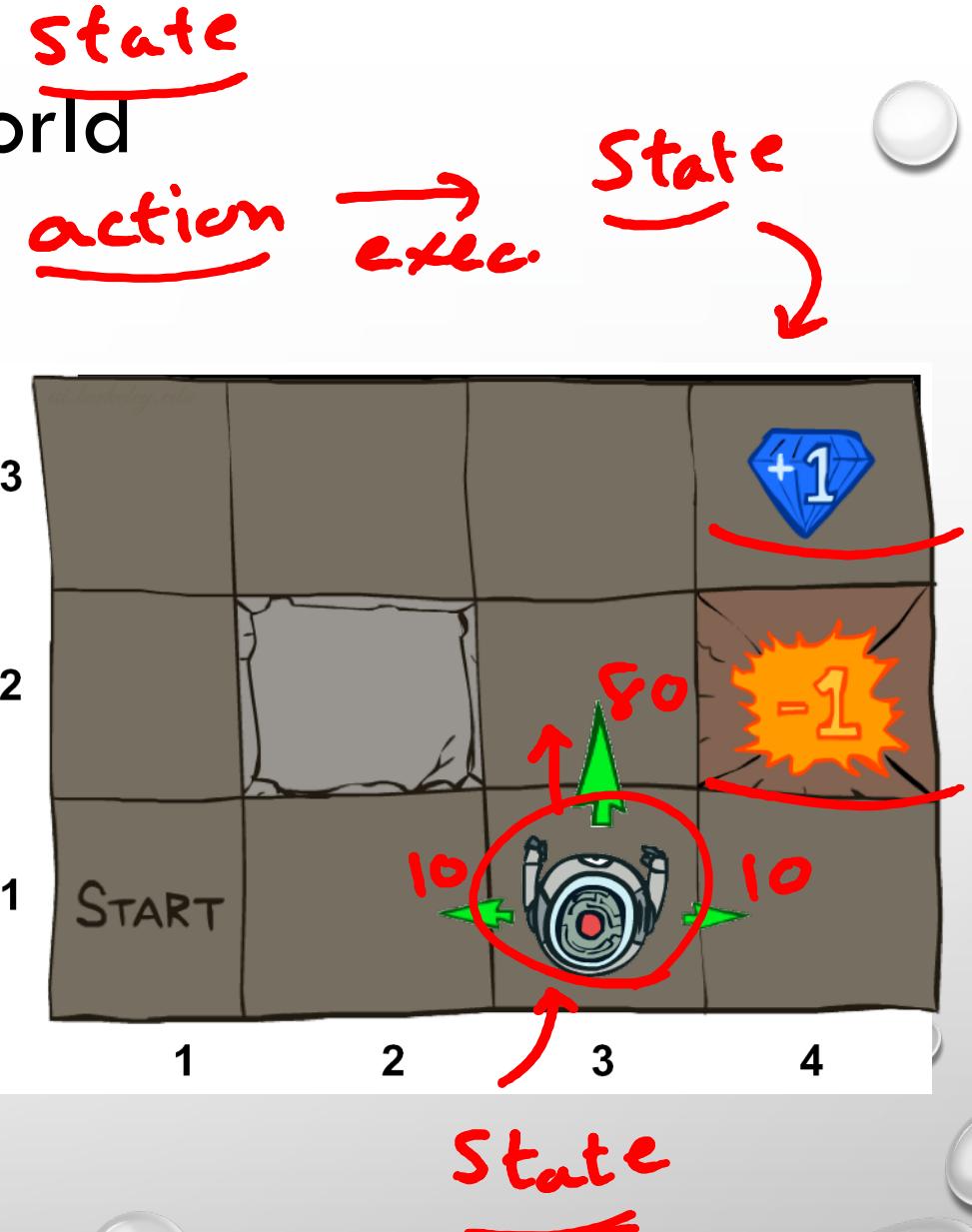
(MDPs)





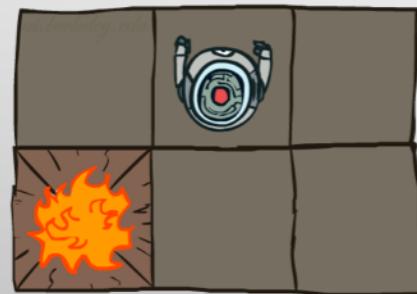
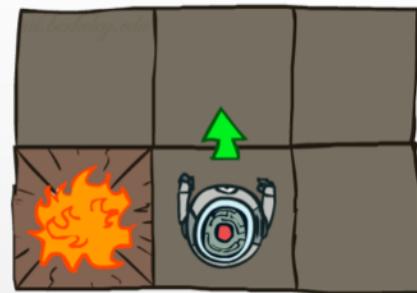
Example: Grid World

- A maze-like problem
 - The agent lives in a grid
 - Walls block the agent's path
- Noisy movement: actions do not always go as planned
 - 80% of the time, the action North takes the agent North (if there is no wall there)
 - 10% of the time, North takes the agent West; 10% East
 - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
 - Small "living" reward each step (can be negative)
 - Big rewards come at the end (good or bad)
- Goal: $\pi: S \rightarrow A$

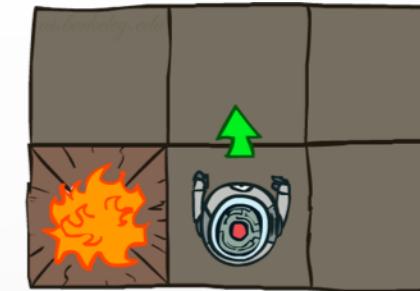


Grid World Actions

- Deterministic Grid World

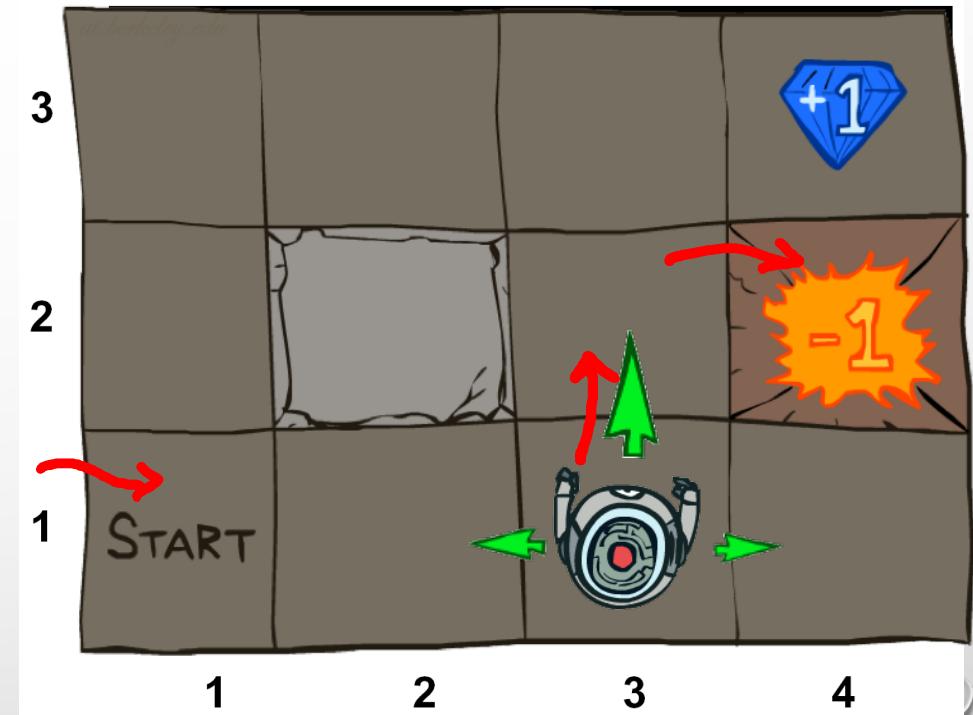


- Stochastic Grid World



Markov Decision Processes

- An MDP is defined by:
 - A set of states $s \in S$
 - A set of actions $a \in A$
 - A transition function $T(s, a, s')$
 - Probability that a from s leads to s' , i.e., $P(s' | s, a)$
 - Also called the model or the dynamics
 - A reward function $R(s, a, s')$
 - Sometimes just $R(s)$ or $R(s')$
 - A start state
 - Maybe a terminal state
- MDPs are non-deterministic search problems
 - One way to solve them is with expectimax search
 - We'll have a new tool soon



What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent
- For Markov decision processes, “Markov” means action outcomes depend only on the current state

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \dots, S_0 = s_0)$$

=

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t) \quad \text{transition function}$$

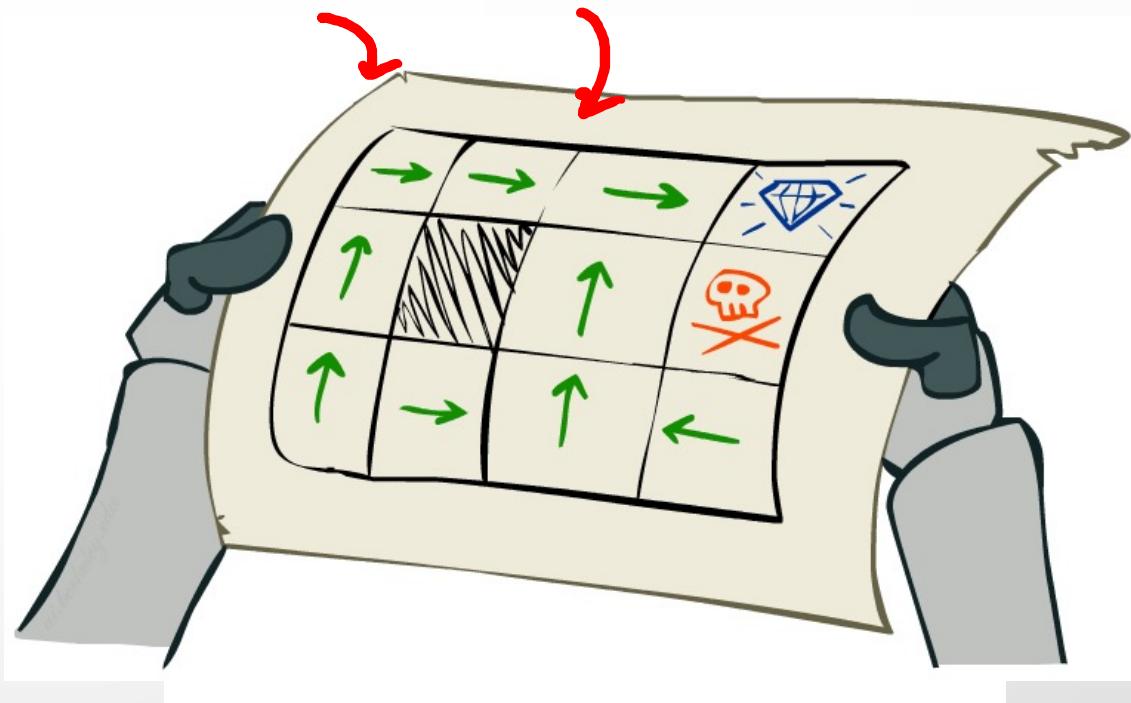
- This is just like search, where the successor function could only depend on the current state (not the history)



Andrey Markov
(1856-1922)

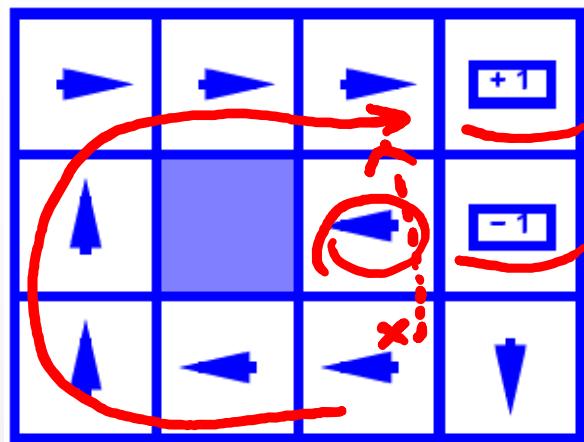
$$\max_{\pi} \mathbb{E} \left\{ \sum_{t=0}^T r(s_t, a_t) \right\}$$

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
 - A policy π gives an action for each state
 - An optimal policy is one that maximizes expected utility if followed
- Expectimax didn't compute entire policies
 - It computed the action for a single state only



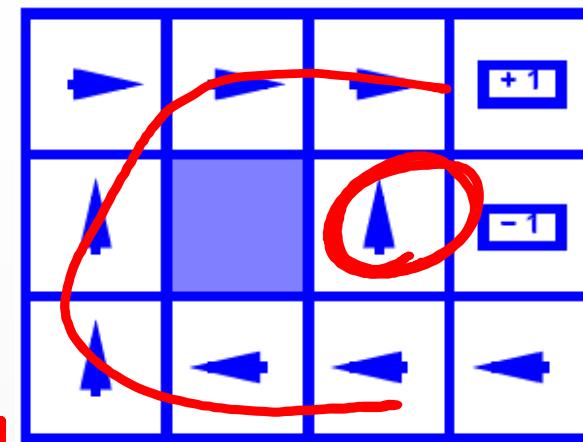
Optimal policy when $R(s, a, s') = -0.03$
for all non-terminals s

Optimal Policies

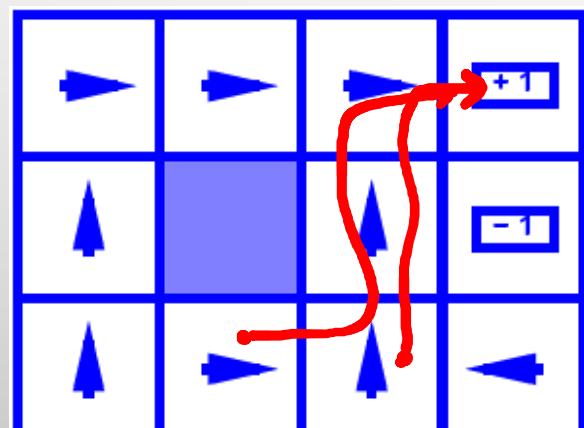


$$R(s) = -0.01$$

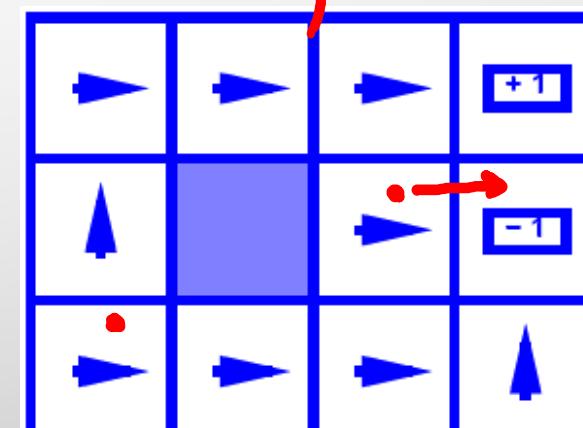
→ living



$$R(s) = -0.03$$

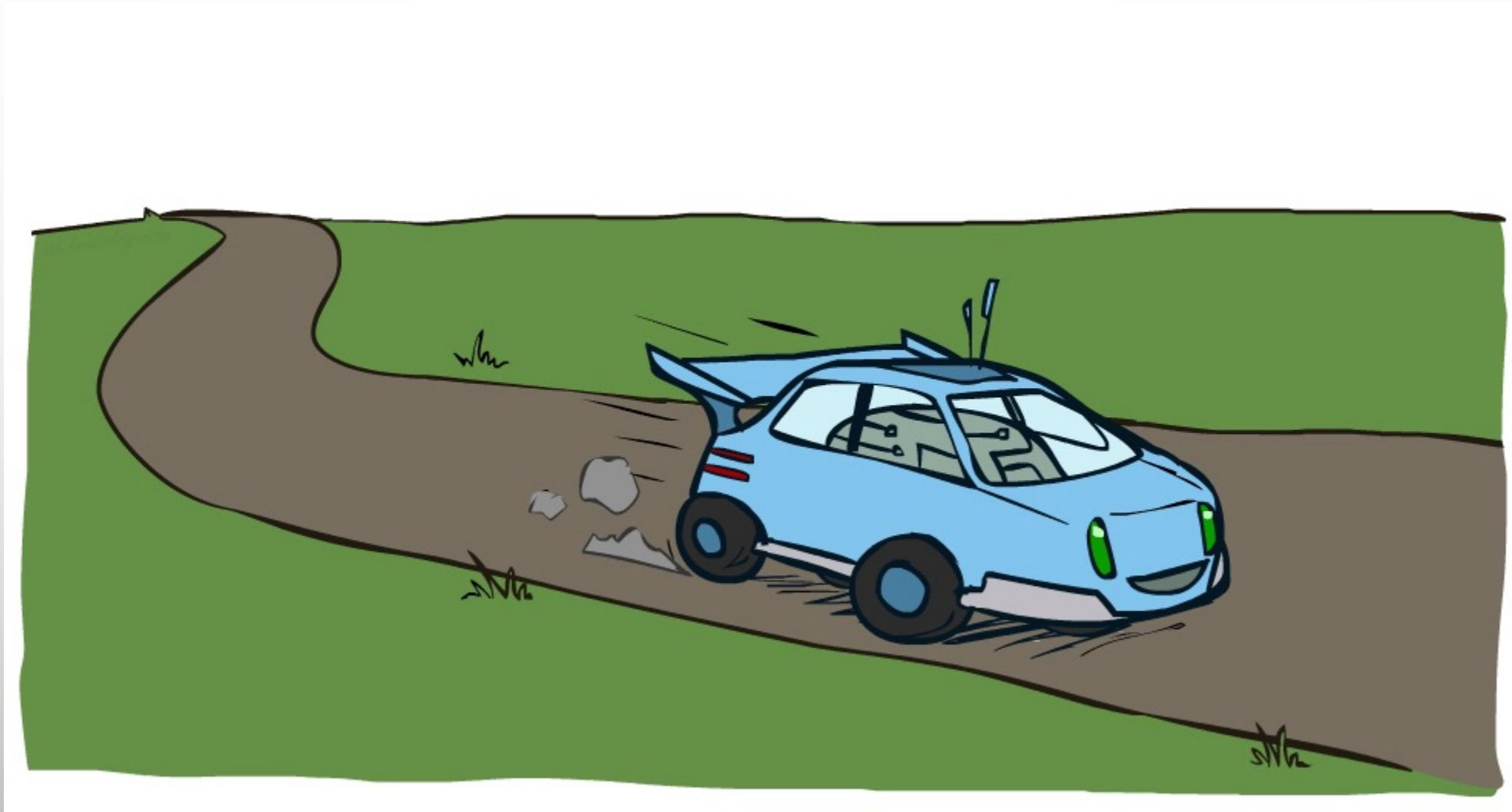


$$R(s) = -0.4$$



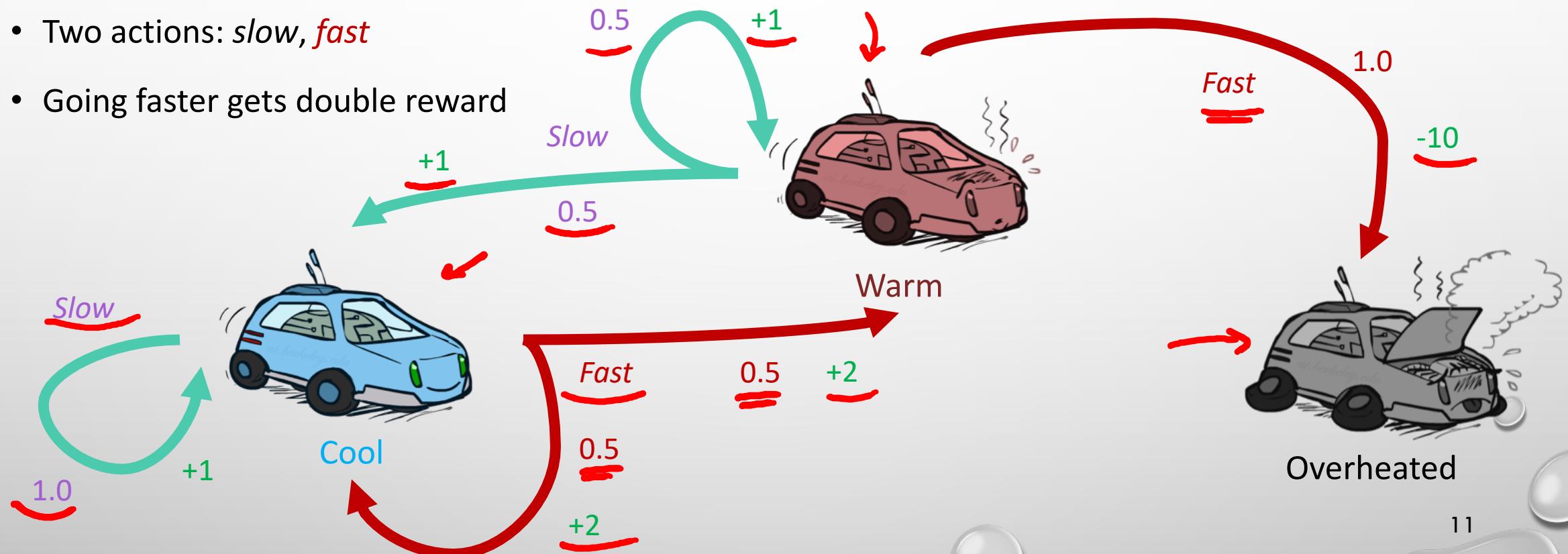
$$R(s) = -2.0$$

Example: Racing

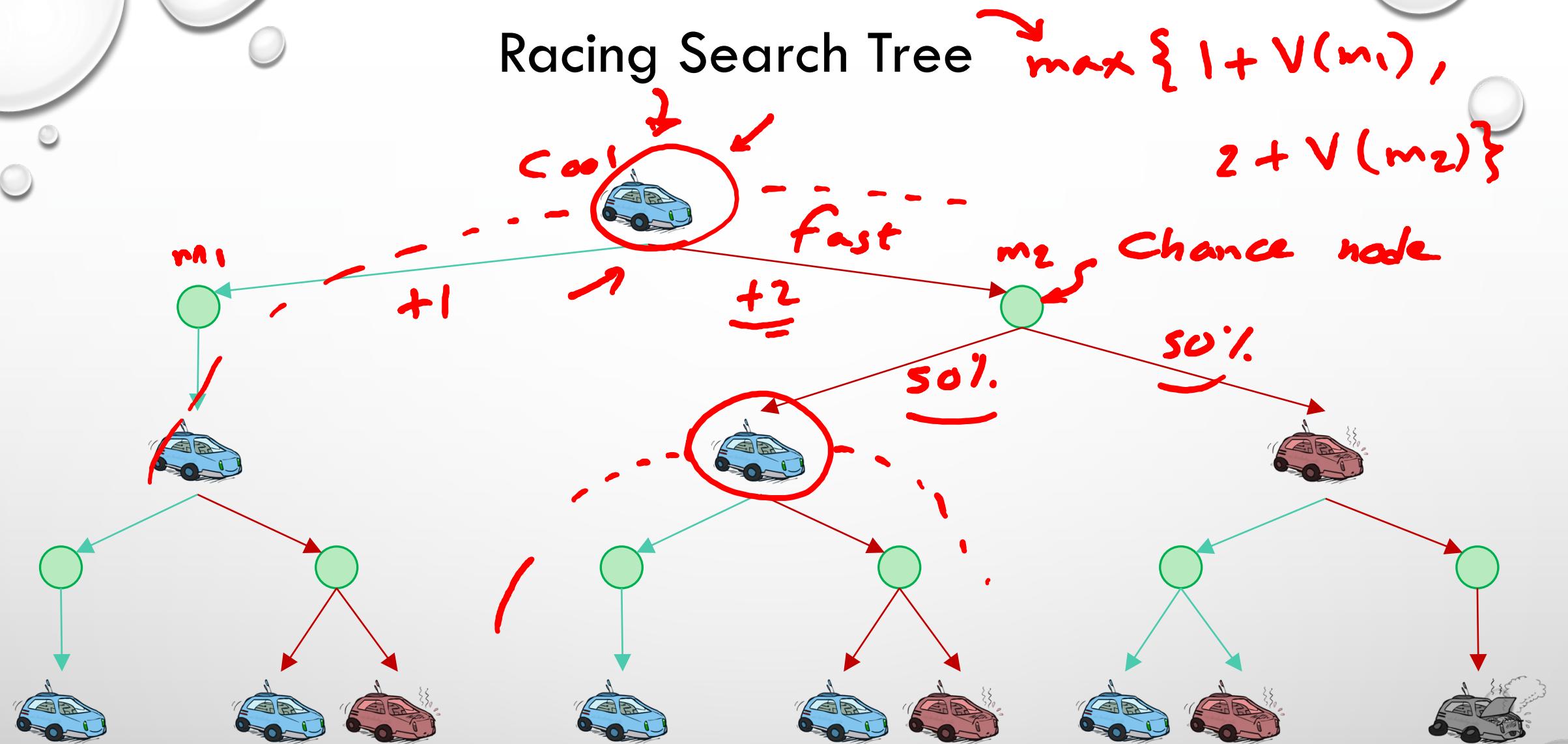


Example: Racing

- A robot car wants to travel far, quickly
- Three states: **cool**, **warm**, overheated
- Two actions: *slow*, *fast*
- Going faster gets double reward

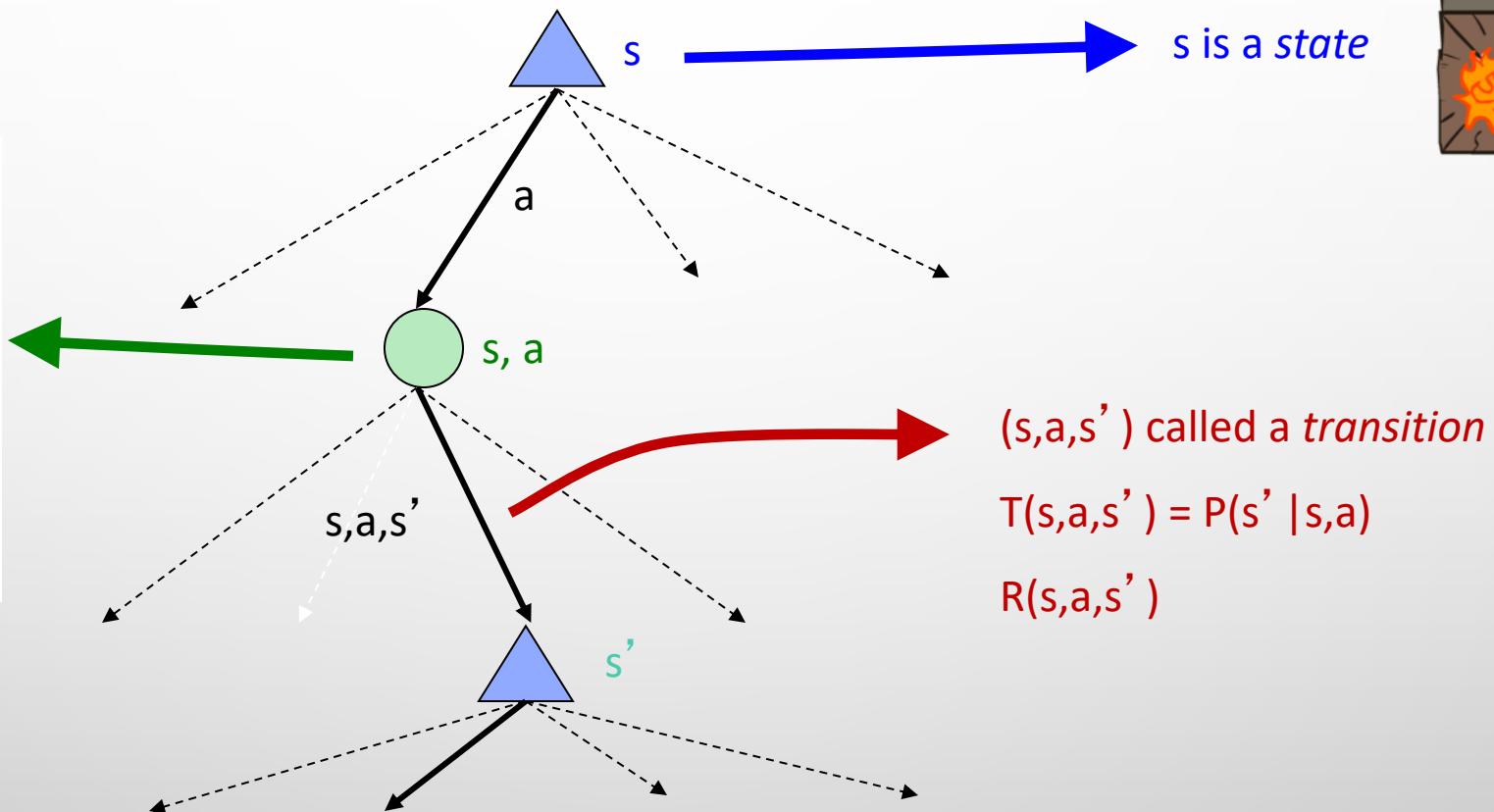


Racing Search Tree

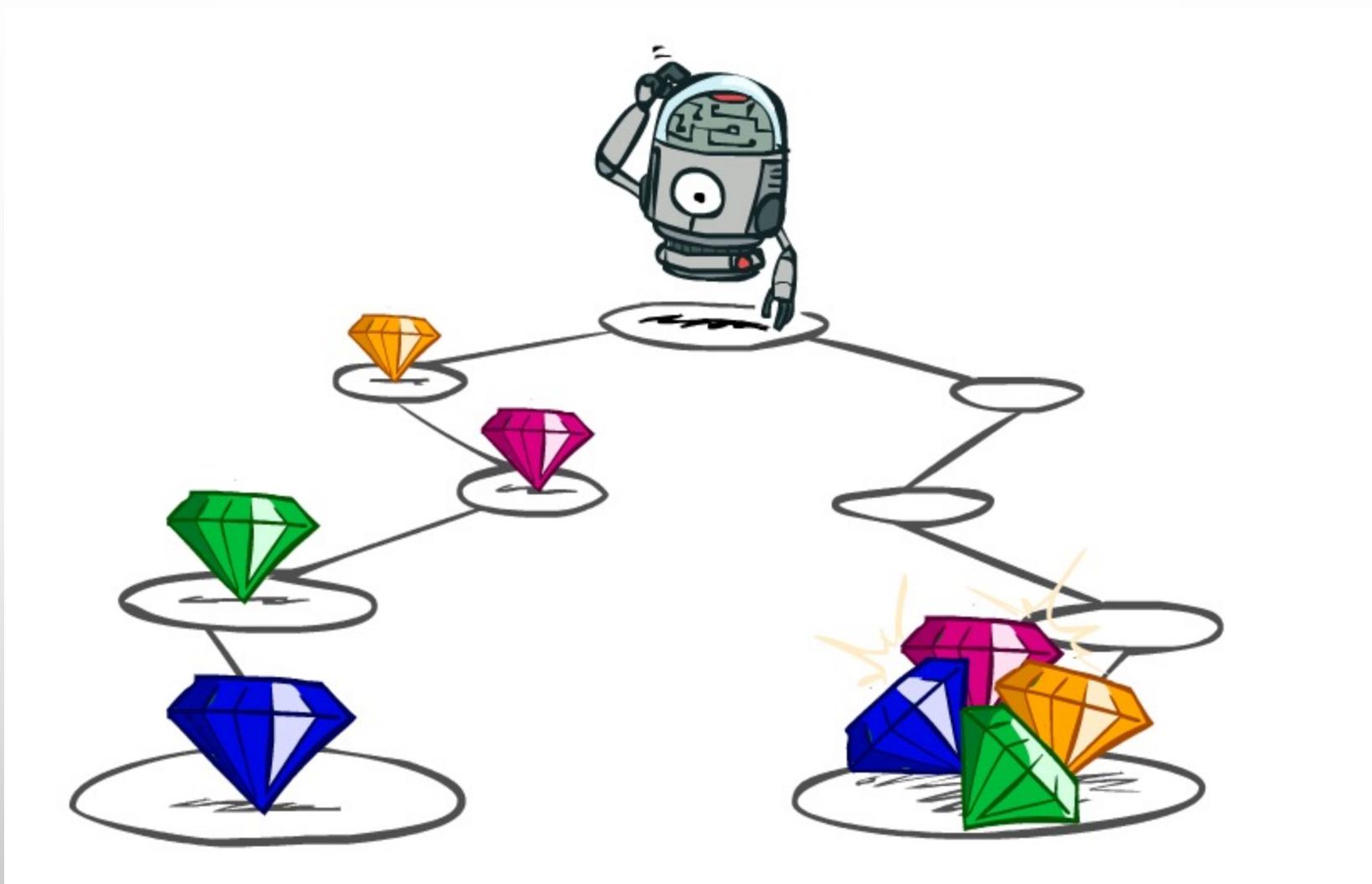


MDP Search Trees

- Each MDP state projects an expectimax-like search tree



Utilities of Sequences



Utilities of Sequences

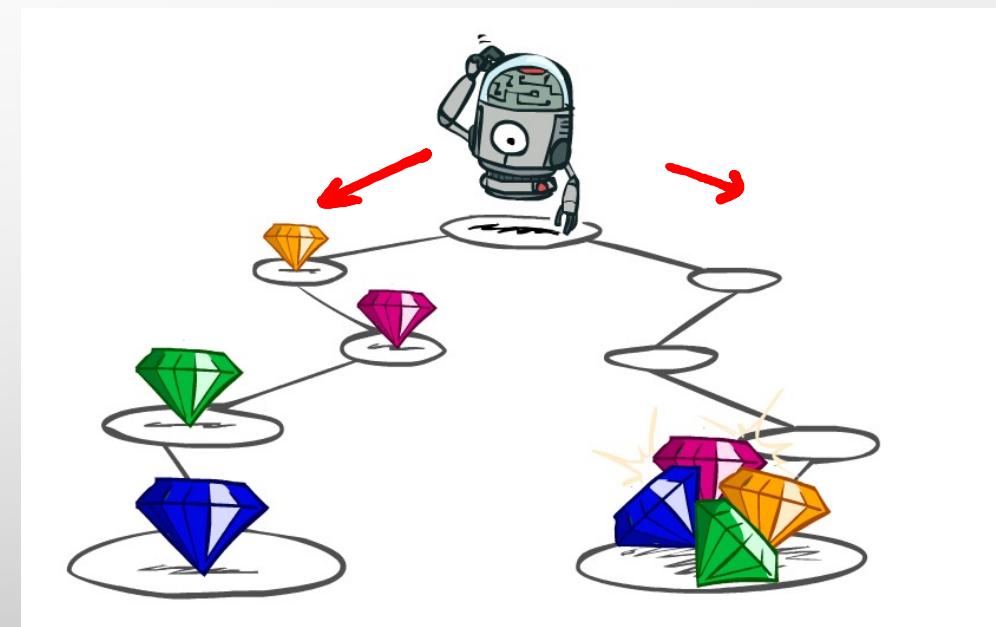
$$\cdot \gamma < 1$$

$$\sum_{t=0}^{\infty} \gamma^t r_t \leq \frac{1}{1-\gamma} \sum_{t=0}^{\infty} r_t$$

- What preferences should an agent have over reward sequences?

- More or less? $[1, 2, 2]$ or $[2, 3, 4]$

- Now or later? $[0, 0, 1]$ or $[1, 0, 0]$



Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially



1

Worth Now



γ

Worth Next Step

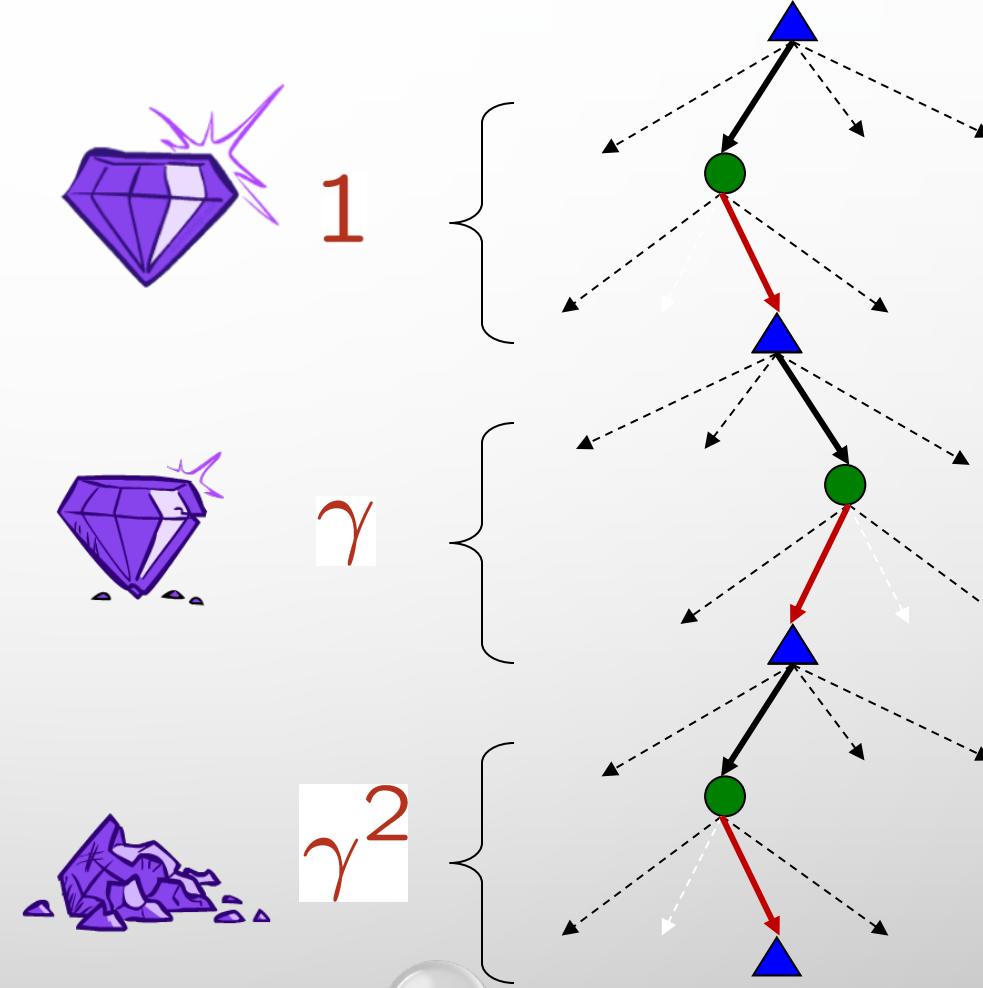


γ^2

Worth In Two Steps

Discounting

- How to discount?
 - Each time we descend a level, we multiply in the discount once
- Why discount?
 - Sooner rewards probably do have higher utility than later rewards
 - Also helps our algorithms converge
- Example: discount of 0.5
 - $U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3$
 - $U([1,2,3]) < U([3,2,1])$



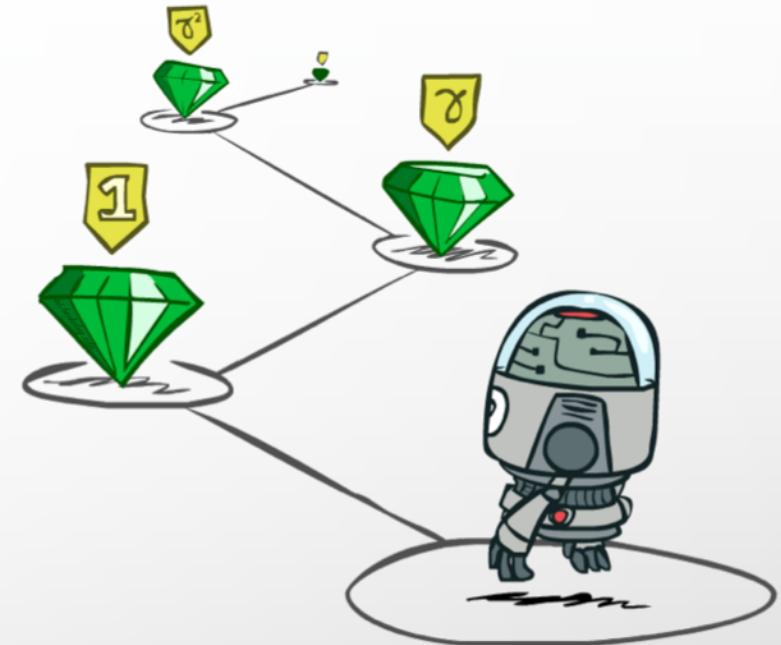
Stationary Preferences

- Theorem: if we assume **stationary preferences**:

$$[a_1, a_2, \dots] \succ [b_1, b_2, \dots]$$

$$\Updownarrow$$

$$[r, a_1, a_2, \dots] \succ [r, b_1, b_2, \dots]$$



- Then: there are only two ways to define utilities

- Additive utility:

$$U([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \dots$$

- Discounted utility:

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \dots$$

Quiz: Discounting

- Given:

10				1
a	b	c	d	e

- Actions: east, west, and exit (only available in exit states a, e)
- Transitions: deterministic
- Quiz 1: for $\gamma = 1$, what is the optimal policy?
- Quiz 2: for $\gamma = 0.1$, what is the optimal policy?
- Quiz 3: for which γ are west and east equally good when in state d?

10				1

10				1

Infinite Utilities?!

- Problem: what if the game lasts forever? Do we get infinite rewards?

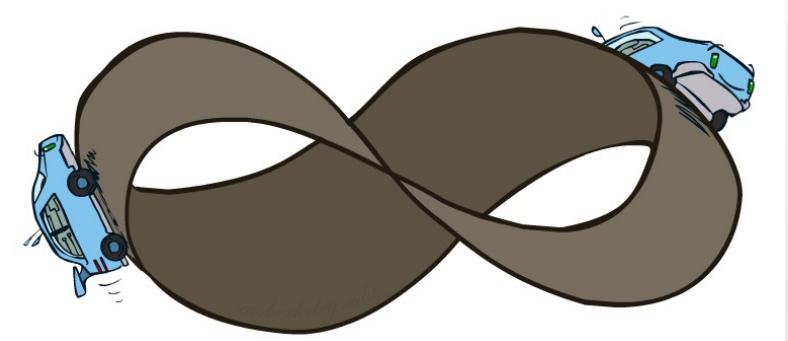
- Solutions:

- Finite horizon: (similar to depth-limited search)
 - Terminate episodes after a fixed T steps (e.g. Life)
 - Gives nonstationary policies (π depends on time left)

- Discounting: use $0 < \gamma < 1$

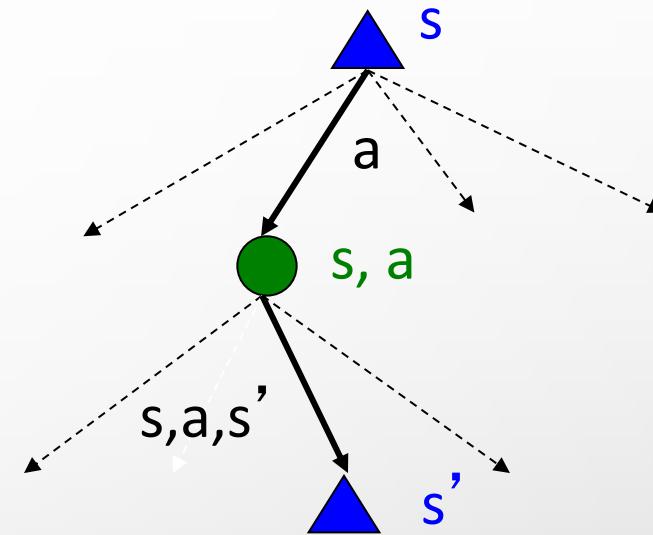
$$U([r_0, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max} / (1 - \gamma)$$

- Smaller γ means smaller “horizon” – shorter term focus
- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

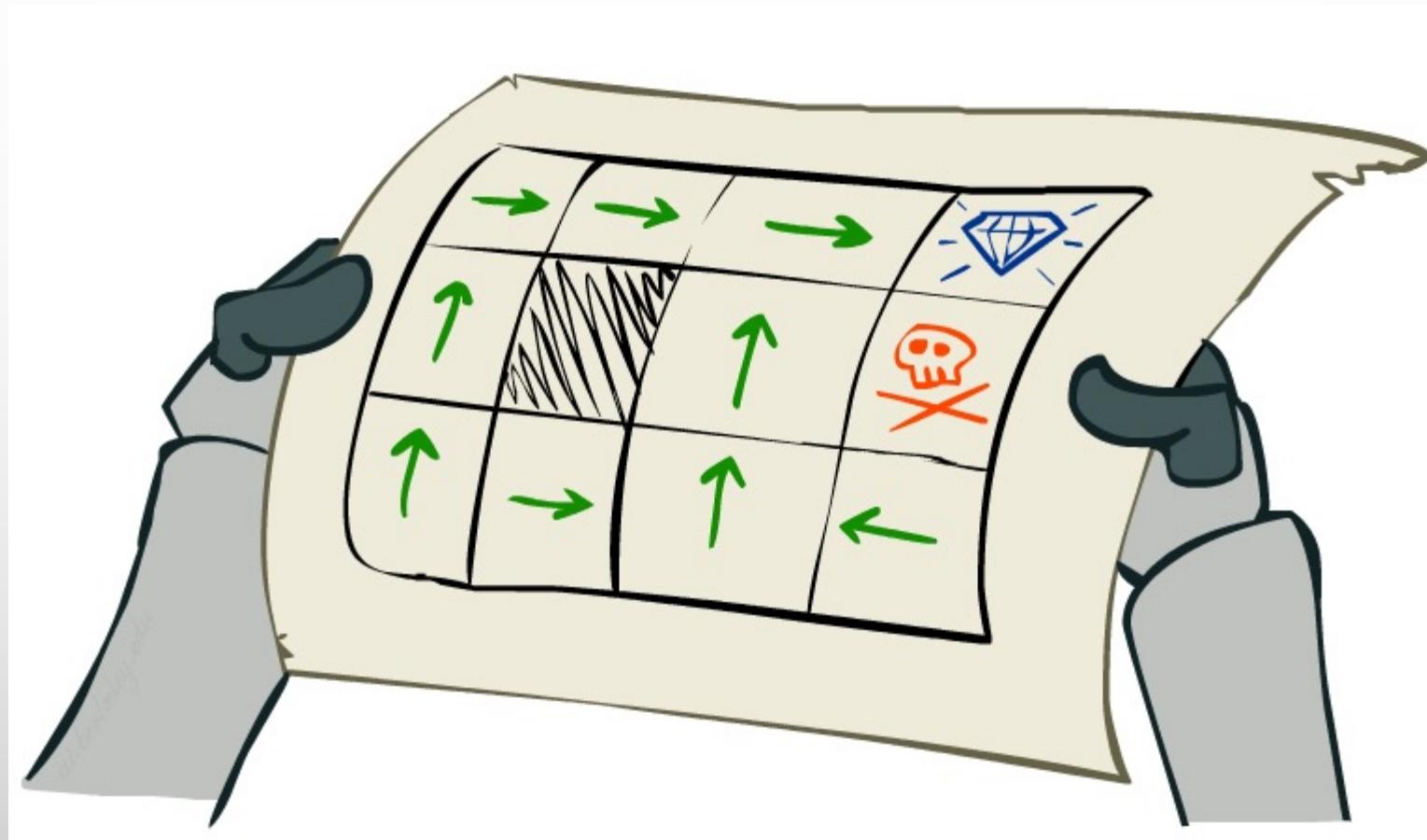


Recap: Defining MDPs

- Markov Decision Processes:
 - Set of states S
 - Start state s_0
 - Set of actions A
 - Transitions $P(s' | s, a)$ (or $T(s, a, s')$)
 - Rewards $R(s, a, s')$ (and discount γ)
- MDP quantities so far:
 - Policy = choice of action for each state
 - Utility = sum of (discounted) rewards



Solving MDPs



Optimal Quantities

Starting @ States s , Sum of rewards when following π^*

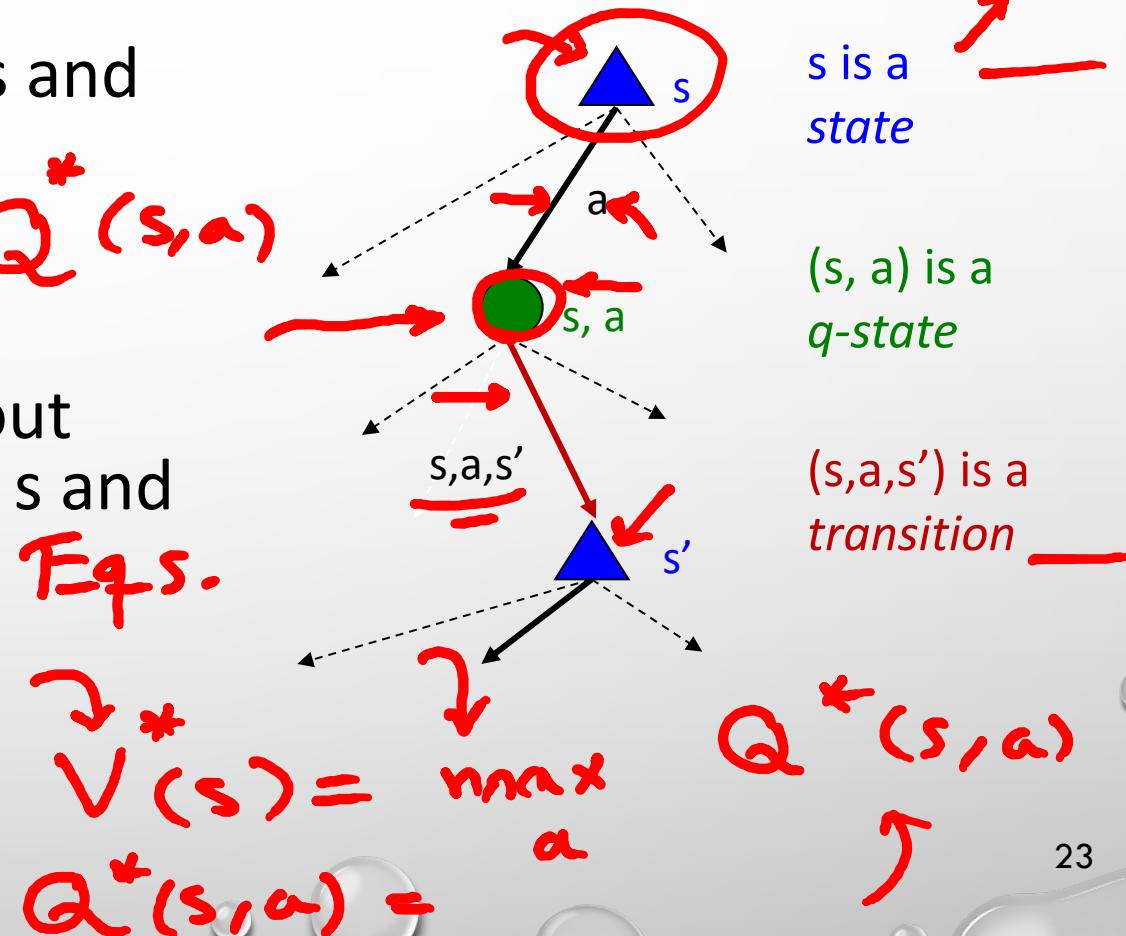
$Q^*(s, a) \rightarrow \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^*(s')]$

- The value (utility) of a state s :
 $V^*(s) = \text{expected utility starting in } s \text{ and acting optimally}$
 $R(s, a, s')$

- The value (utility) of a q-state (s, a) :
 $Q^*(s, a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally}$

Bellman's Optimality Eqs.

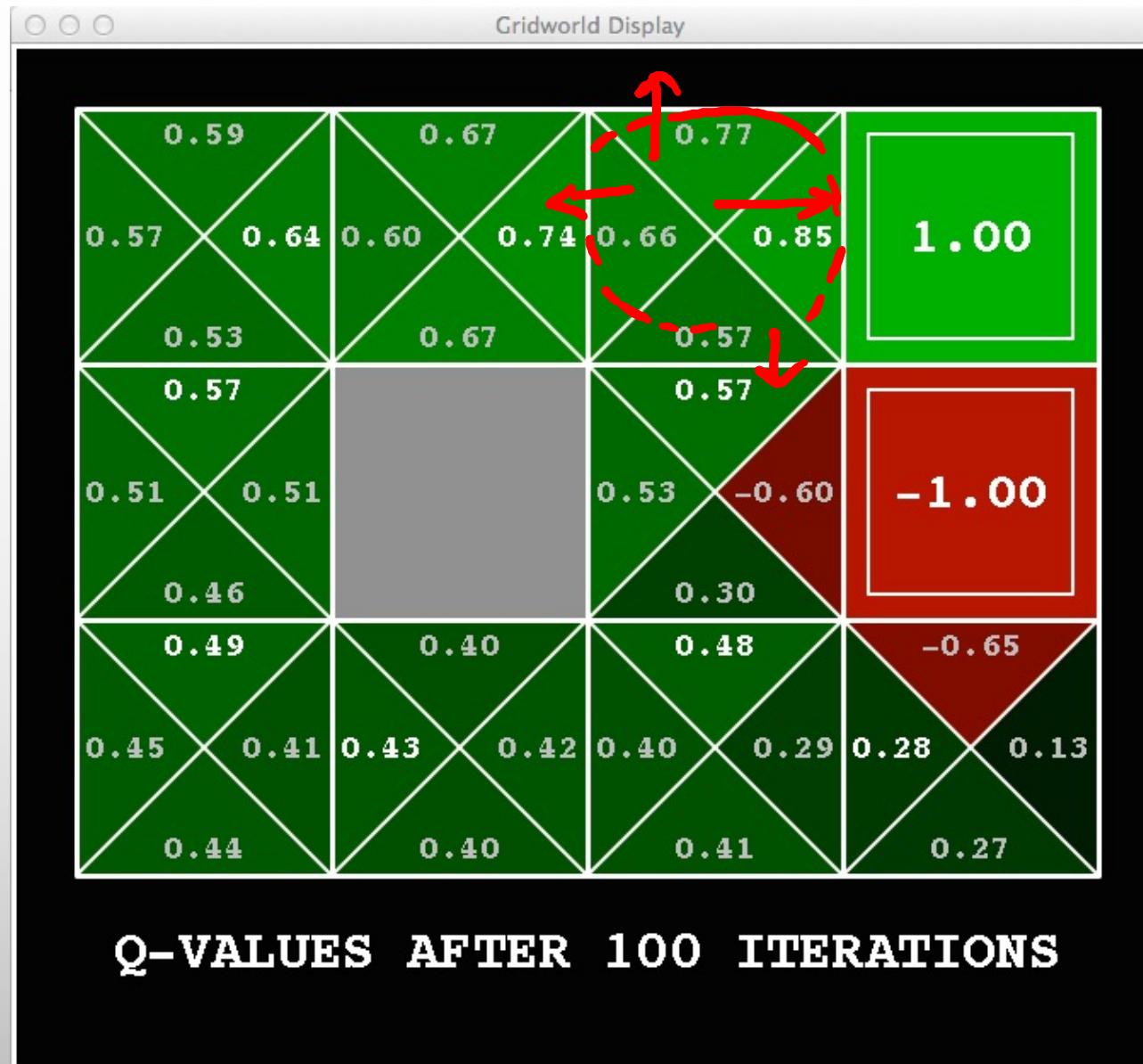
- The optimal policy:
 $\pi^*(s) = \text{optimal action from state } s$



Snapshot of Demo – Gridworld V Values



Snapshot of Demo – Gridworld Q Values



Values of States

- Fundamental operation: compute the (expectimax) value of a state

- Expected utility under optimal action
- Average sum of (discounted) rewards
- This is just what expectimax computed!

$$V^*(s) = \max_a Q^*(s, a)$$
$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

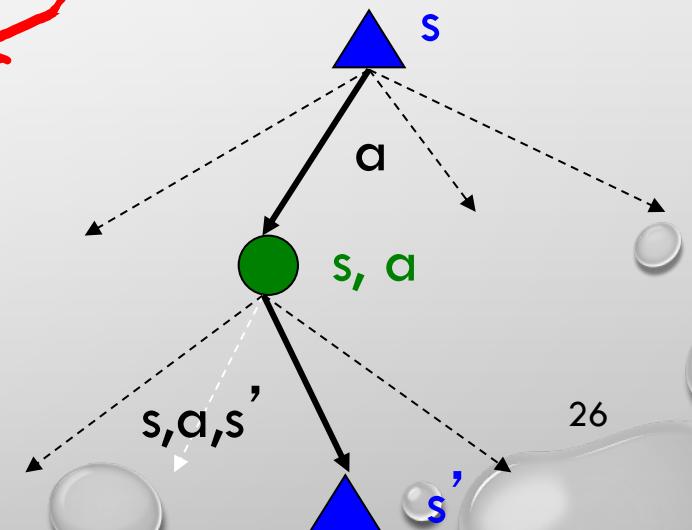
- Recursive definition of value:

W.S.:

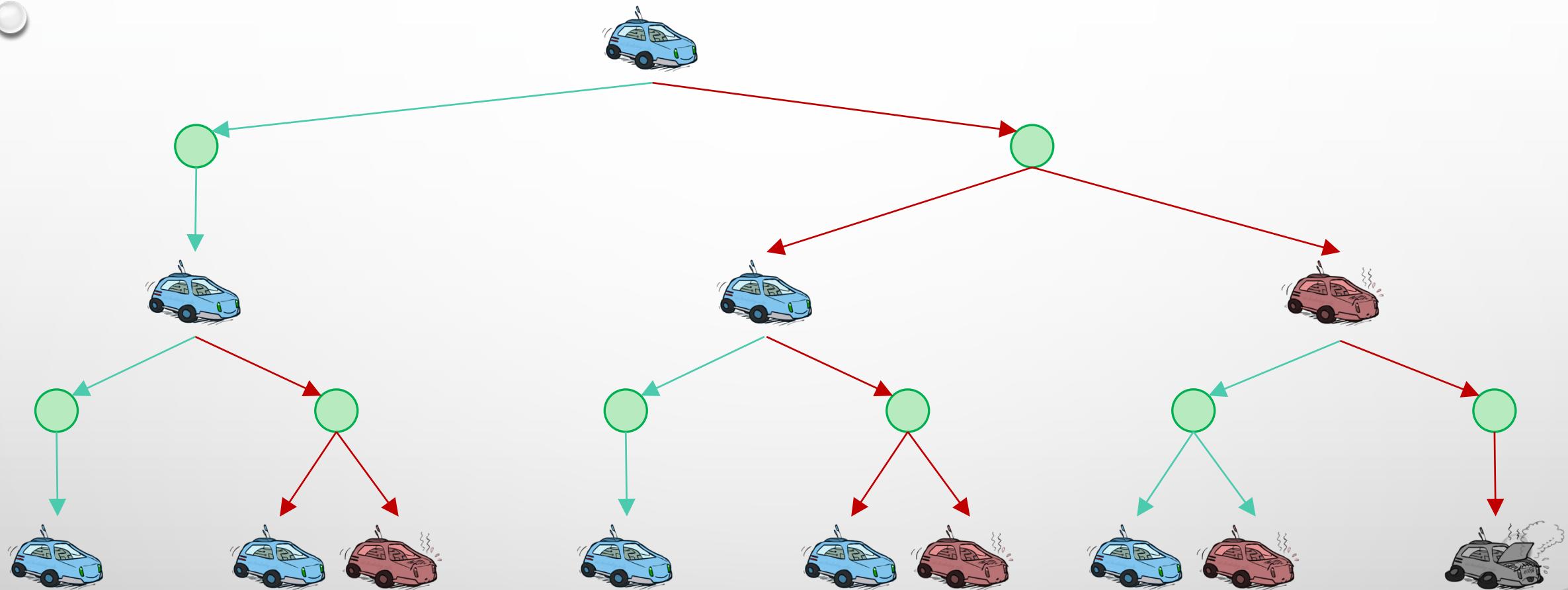
$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

$P(s' | s, a)$

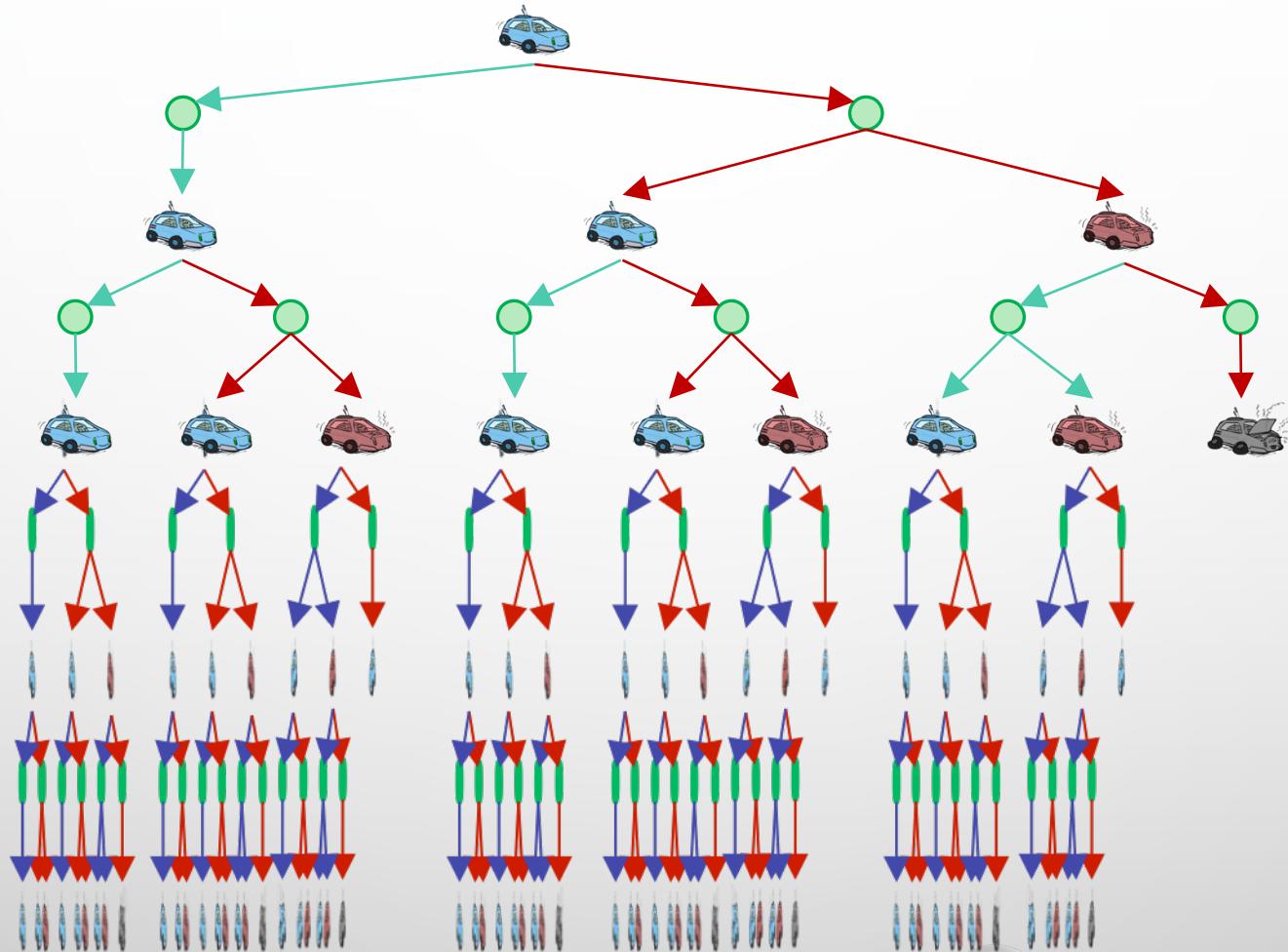
Val. of
the next



Racing Search Tree

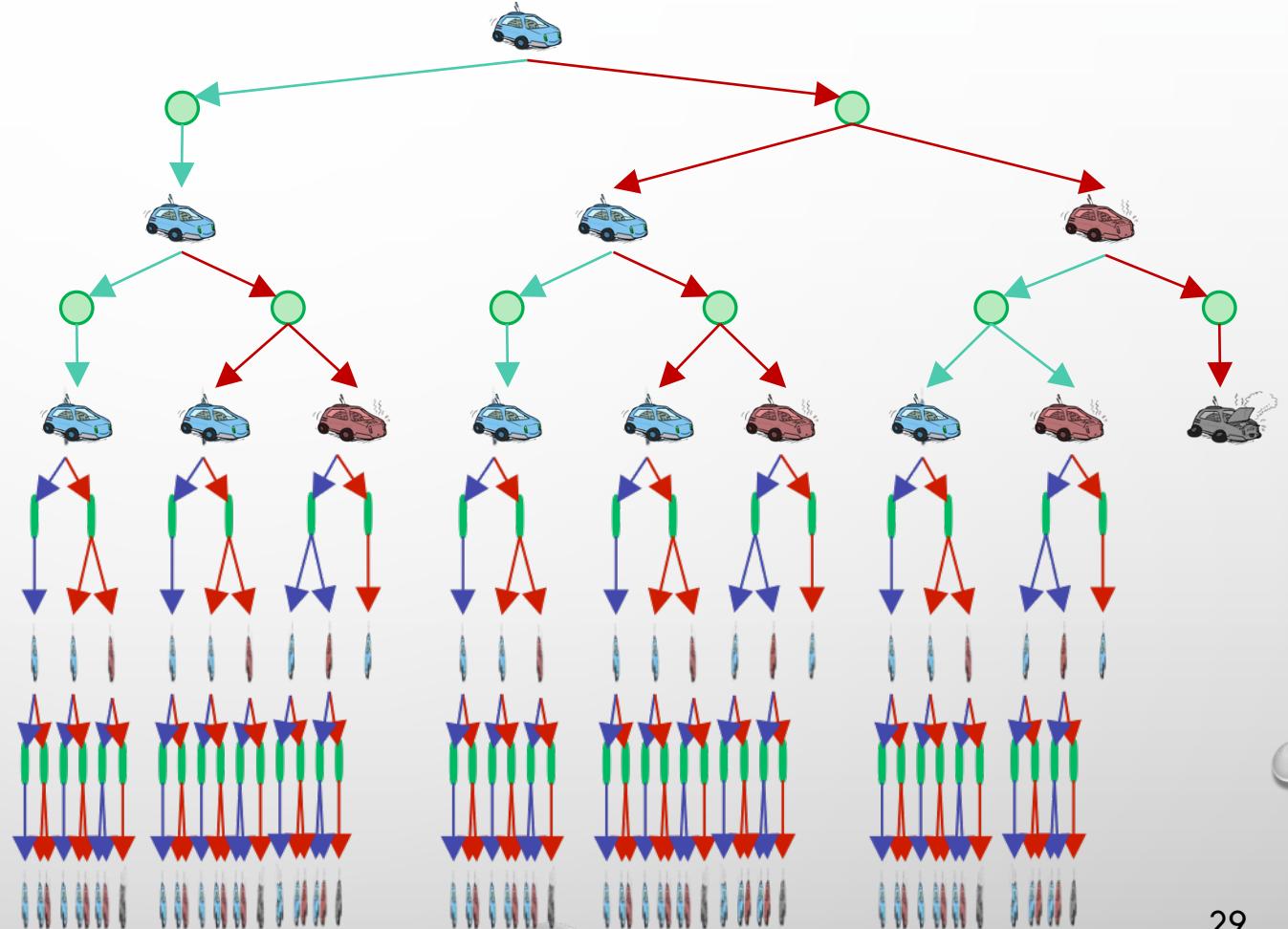


Racing Search Tree



Racing Search Tree

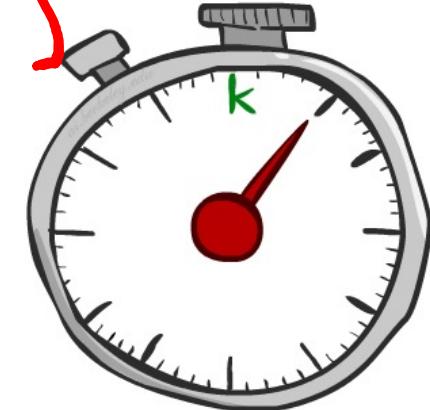
- We're doing way too much work with expectimax!
- Problem: states are repeated
 - Idea: only compute needed quantities once
- Problem: tree goes on forever
 - Idea: do a depth-limited computation, but with increasing depths until change is small
 - Note: deep parts of the tree eventually don't matter if $\gamma < 1$



Bellman Operator Time-Limited Values

- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
 - Equivalently, it's what a depth- k expectimax would give from s

$$V_0 = \{ V(s_1), \dots, V(s_k) \}$$



$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

$V_k(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

$\lim_{k \rightarrow \infty} V_k = V^*$

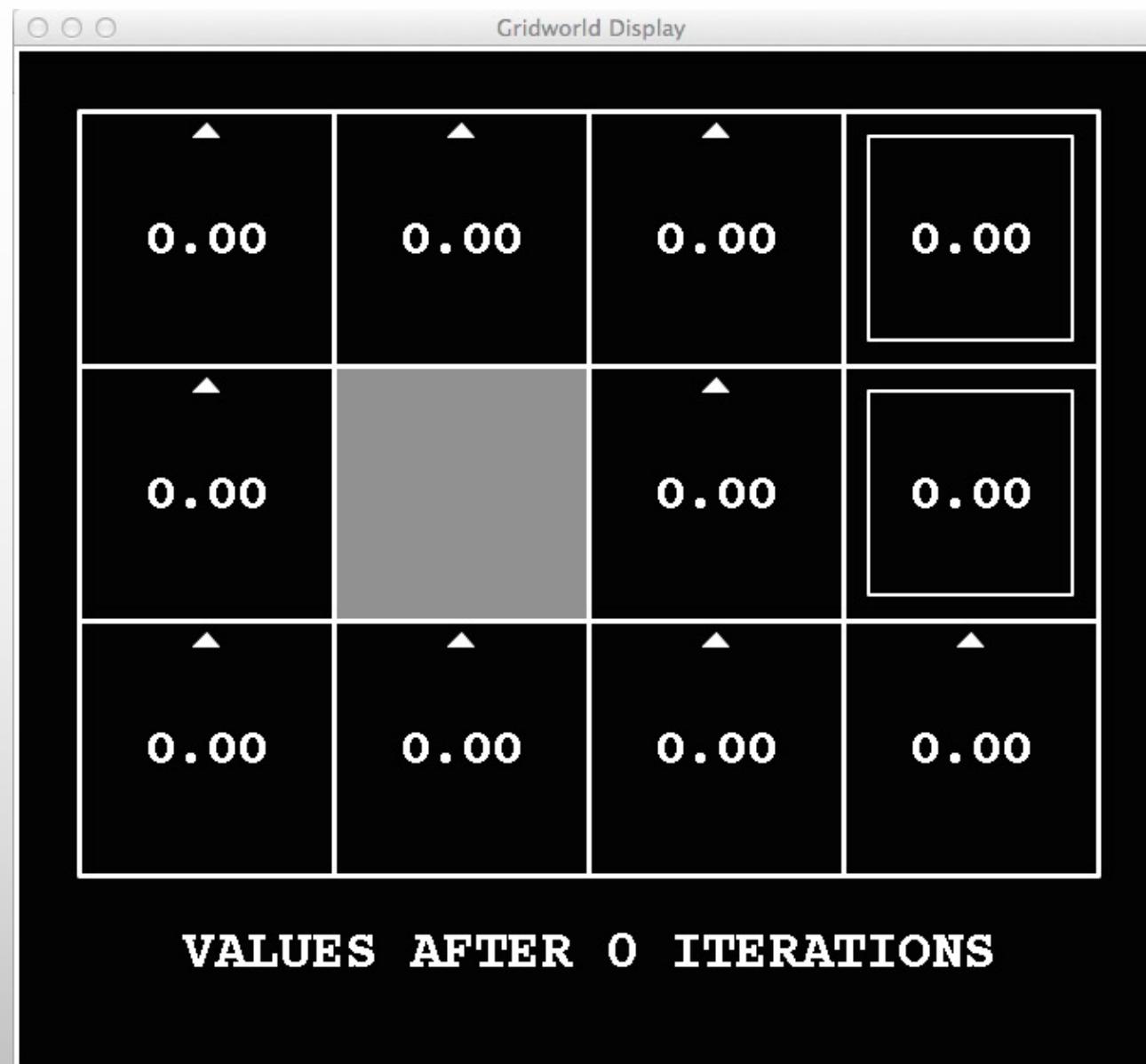
Val. Iteration

fixed point iteration

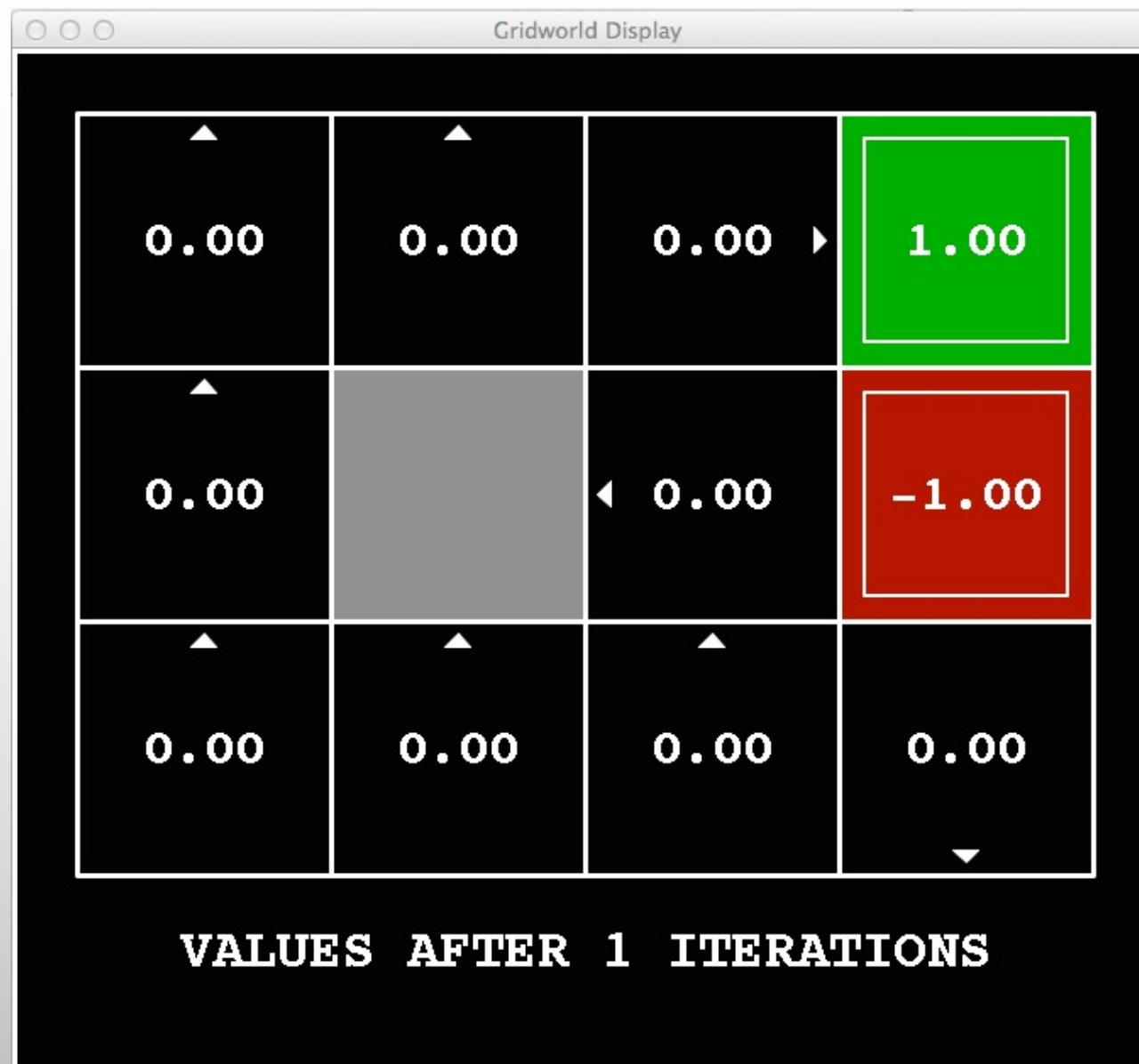
$V_2(\text{car})$

$V_{k-1}^{*}(s')$

$k=0$



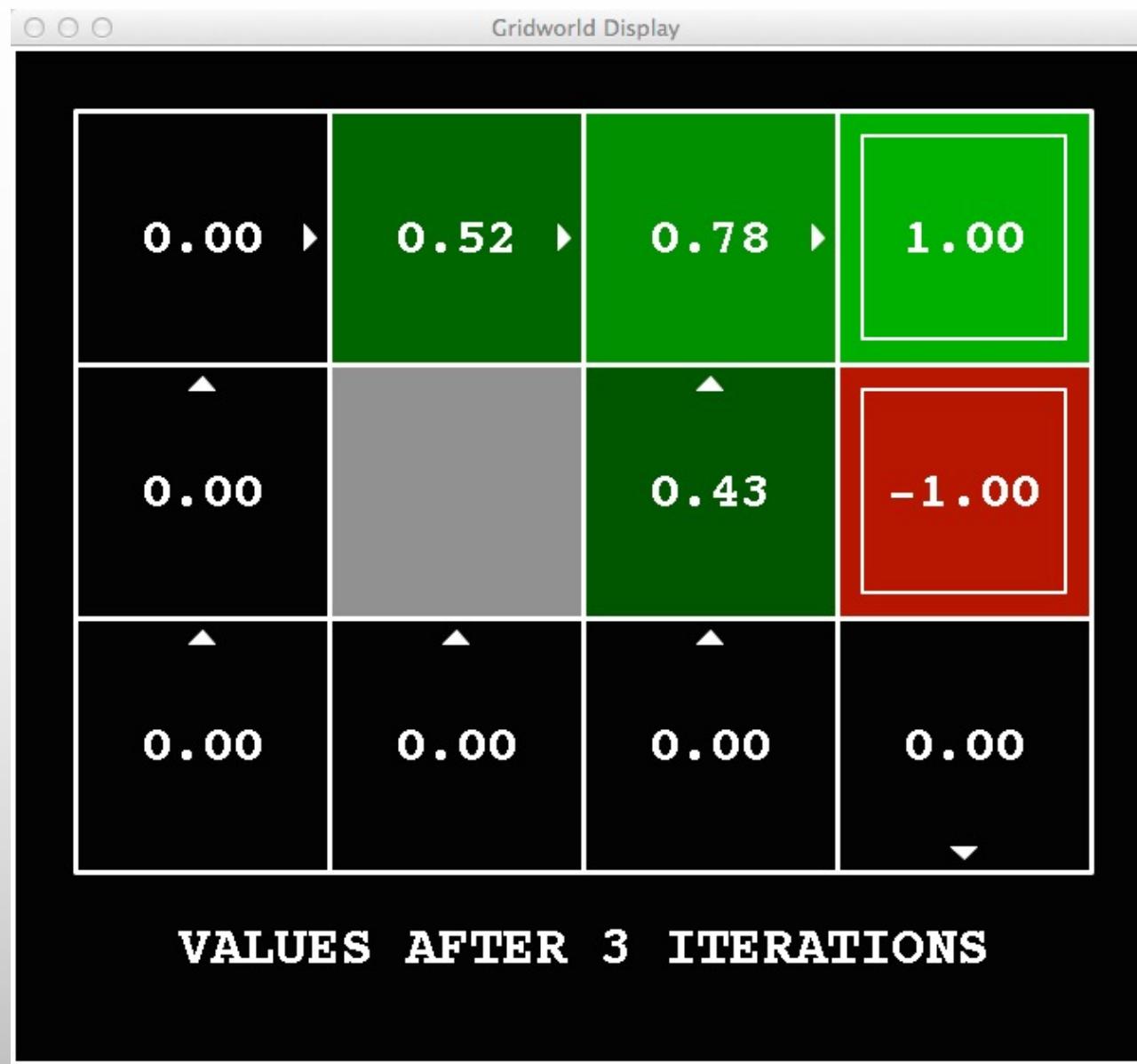
$k=1$



$k=2$



k=3



k=4



k=5



k=6



VALUES AFTER 6 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



k=10



Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



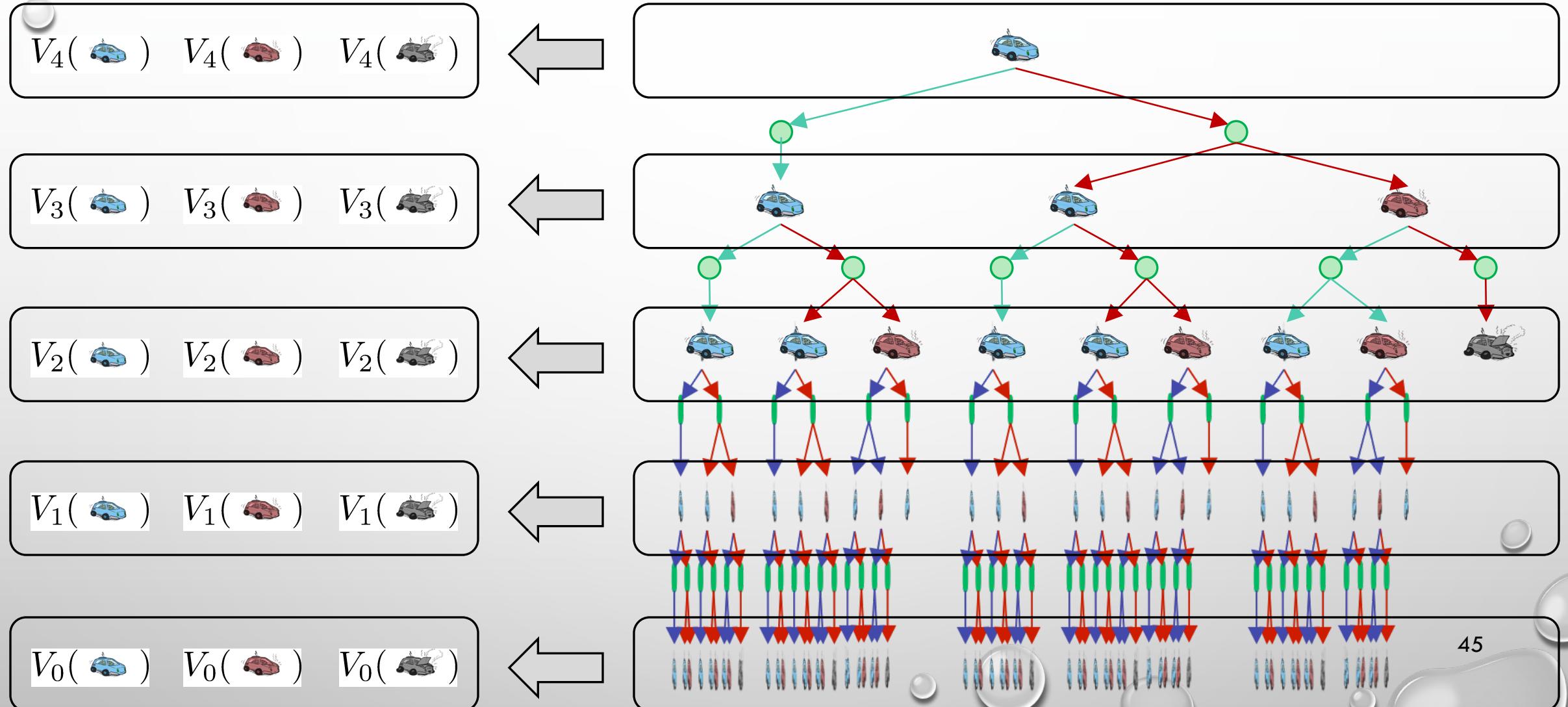
k=12



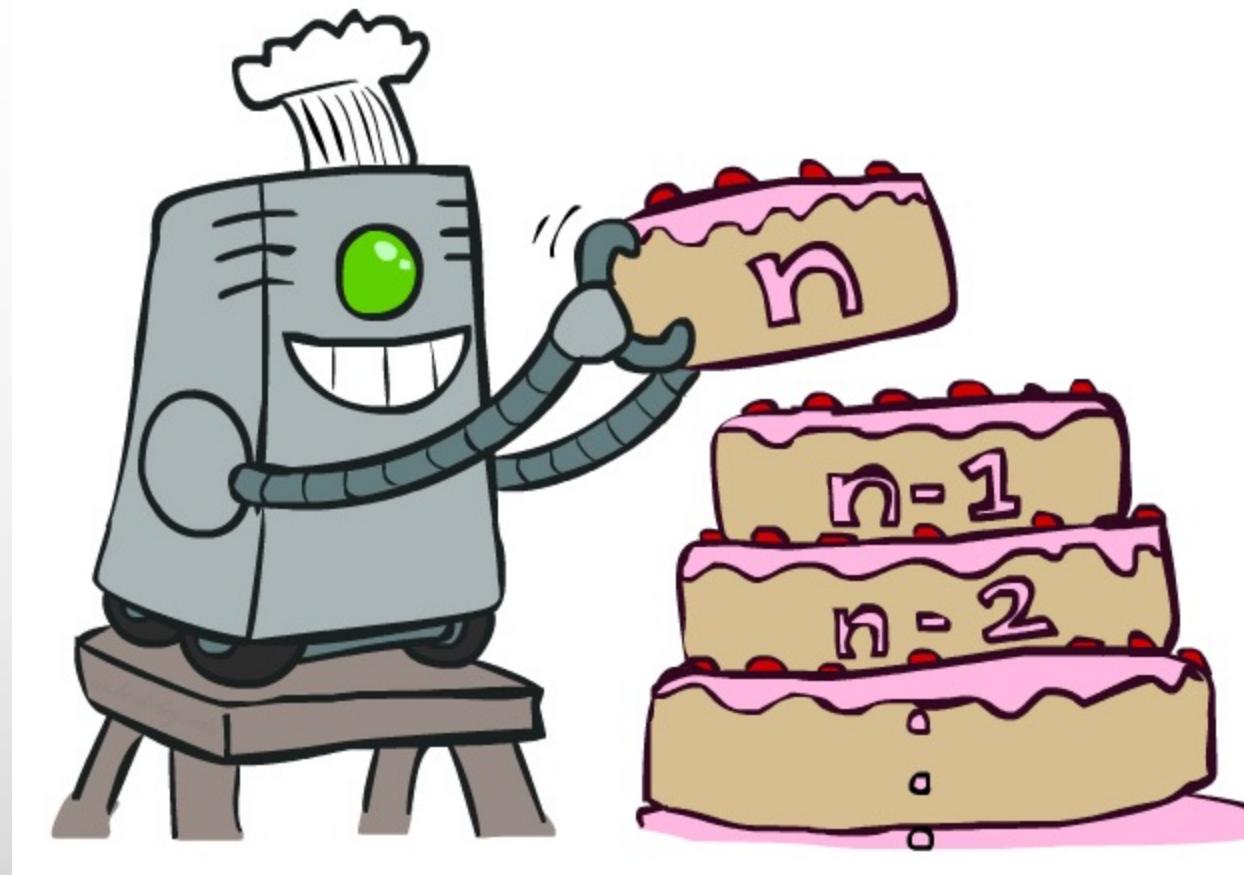
k=100



Computing Time-Limited Values



Value Iteration

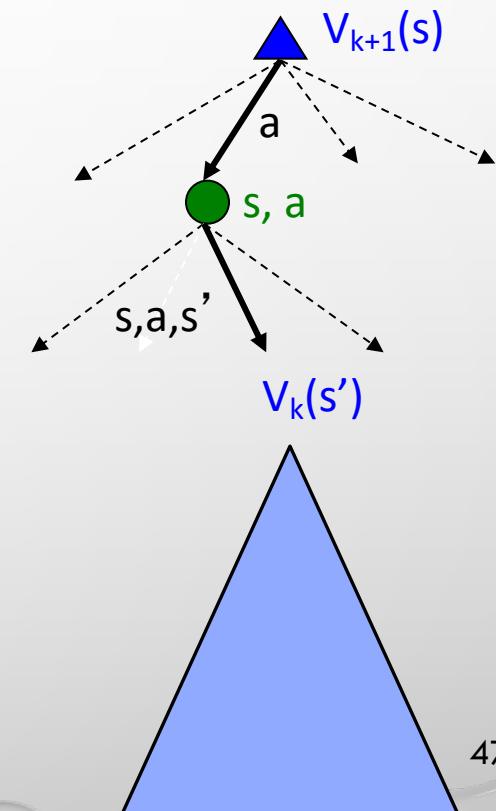


Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

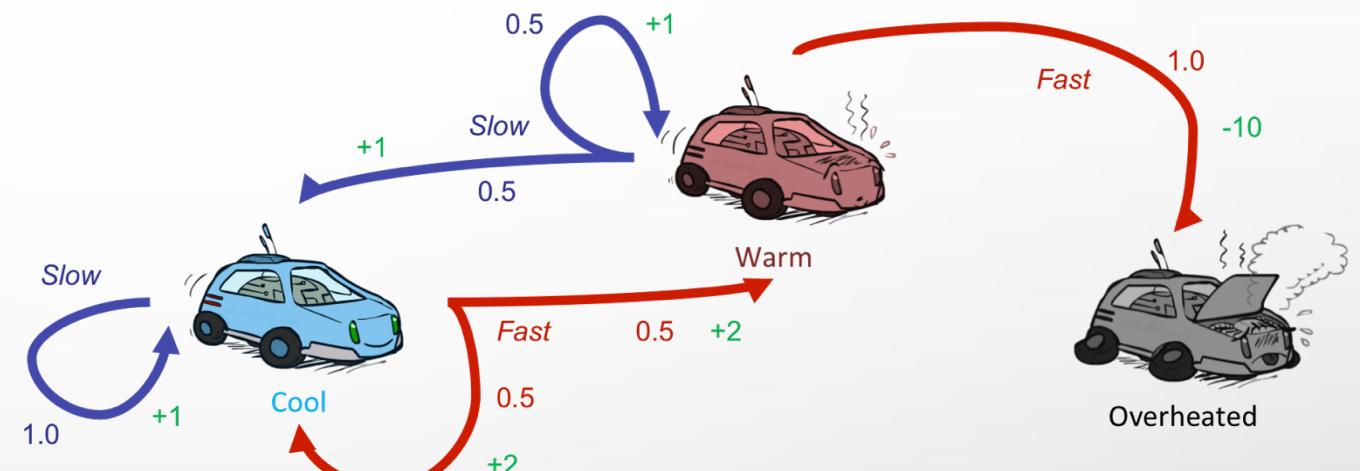
- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
 - Basic idea: approximations get refined towards optimal values
 - Policy may converge long before values do



Example: Value Iteration

			
V_2	3.5	2.5	0
	<u>fast</u>	<u>Slow</u>	
V_1	2	1	0
	<u>fast</u>	<u>Slow</u>	
V_0	0	0	0

$$V_1(\text{cool}) = \{ 1 + V_0(\text{cool}), 2 + 0.5 V_0(\text{cool}) + 0.5 V_0(\text{warm}) \}$$



$$\gamma = 1$$

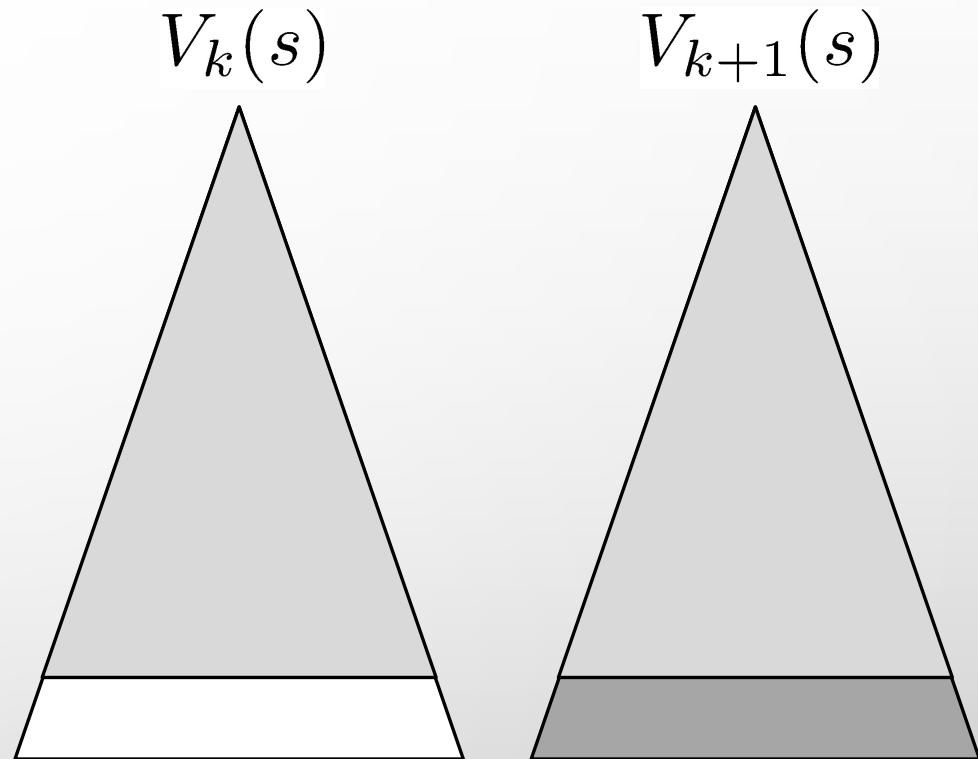
Assume no discount!

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

$$V_1(\text{cool}) = \{ 1 + V_0(\text{cool}), 2 + 0.5 V_0(\text{cool}) + 0.5 V_0(\text{warm}) \}$$

Convergence

- How do we know the V_k vectors are going to converge?
- Case 1: if the tree has maximum depth M , then V_M holds the actual untruncated values
- Case 2: if the discount is less than 1
 - Sketch: for any state V_k and V_{k+1} can be viewed as depth $k+1$ expectimax results in nearly identical search trees
 - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
 - That last layer is at best all R_{MAX}
 - It is at worst R_{MIN}
 - But everything is discounted by γ^k that far out
 - So V_k and V_{k+1} are at most $\gamma^k \max |R|$ different
 - So as k increases, the values converge



Next: Policy-Based Methods

Eval.

π_0

$V_{\pi_0}(s)$

The Bellman Equations

π_1
policy Improv.

Policy Iteration

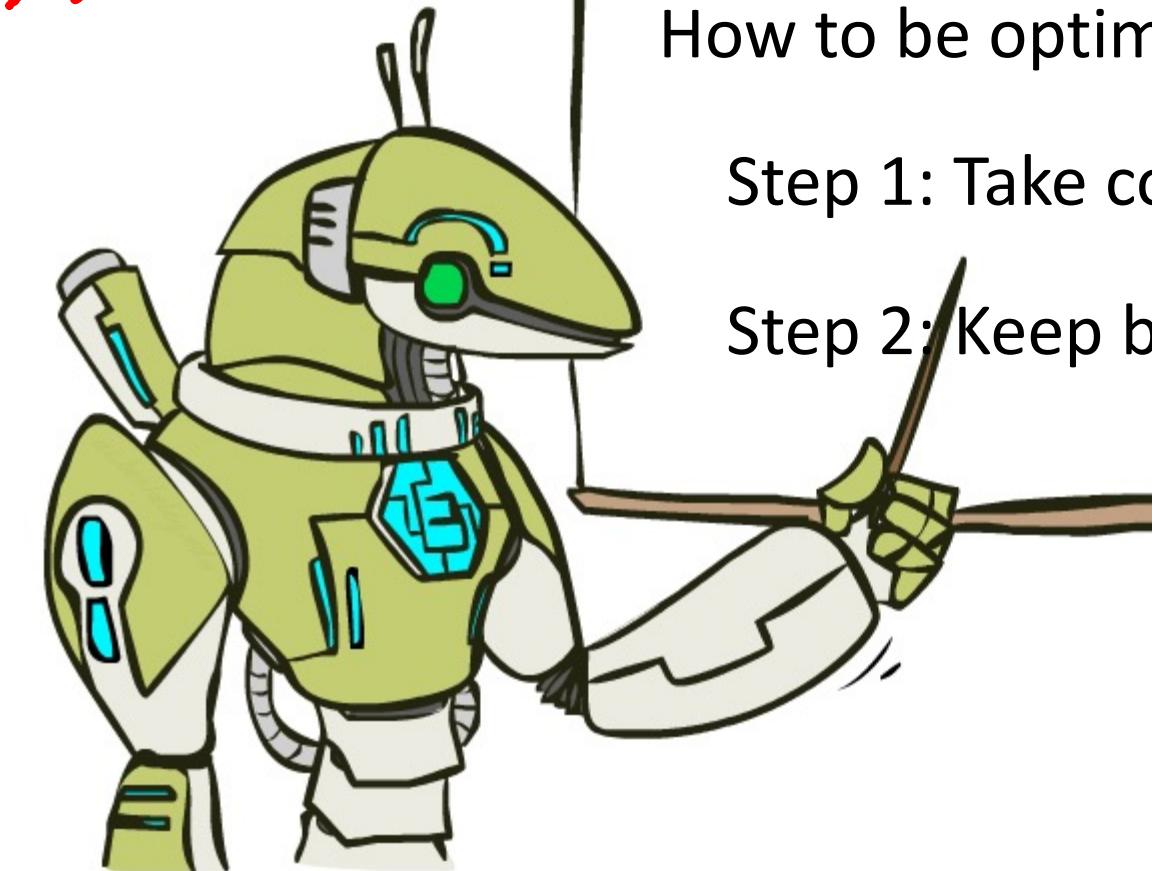
$V_{\pi_1}(s) \rightarrow \pi_2$

→

How to be optimal:

Step 1: Take correct first action

Step 2: Keep being optimal



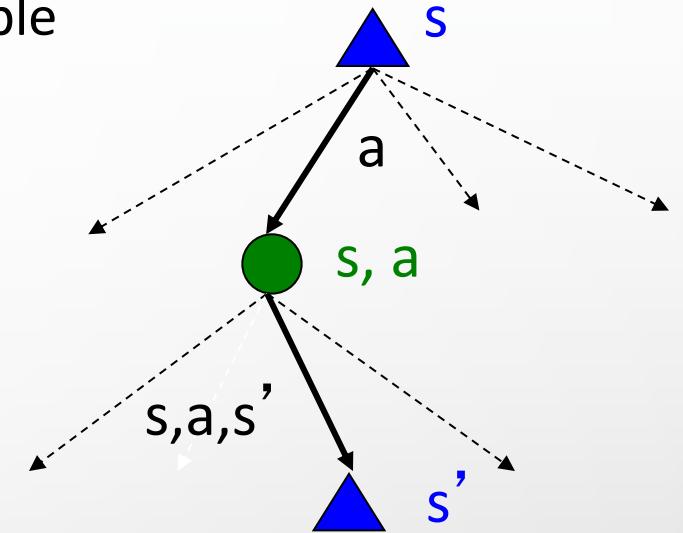
The Bellman Equations

- Definition of “optimal utility” via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

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

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$



- These are the bellman equations, and they characterize optimal values in a way we'll use over and over

Value Iteration

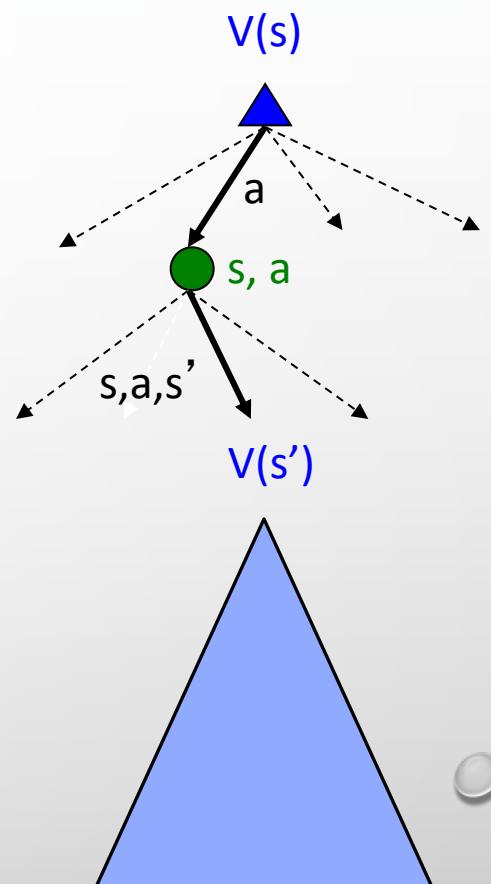
- Bellman equations **characterize** the optimal values:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

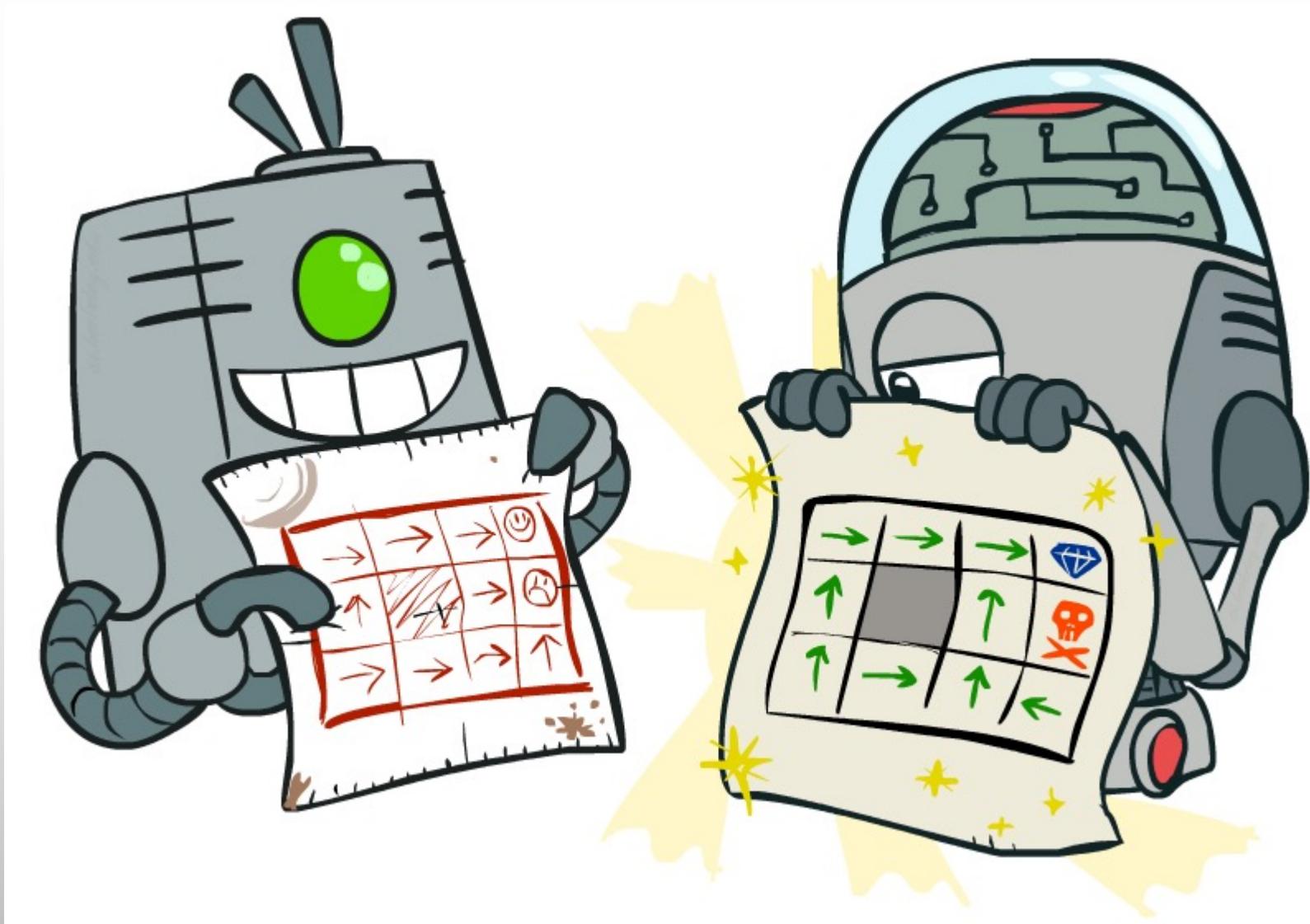
- Value iteration **computes** them:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

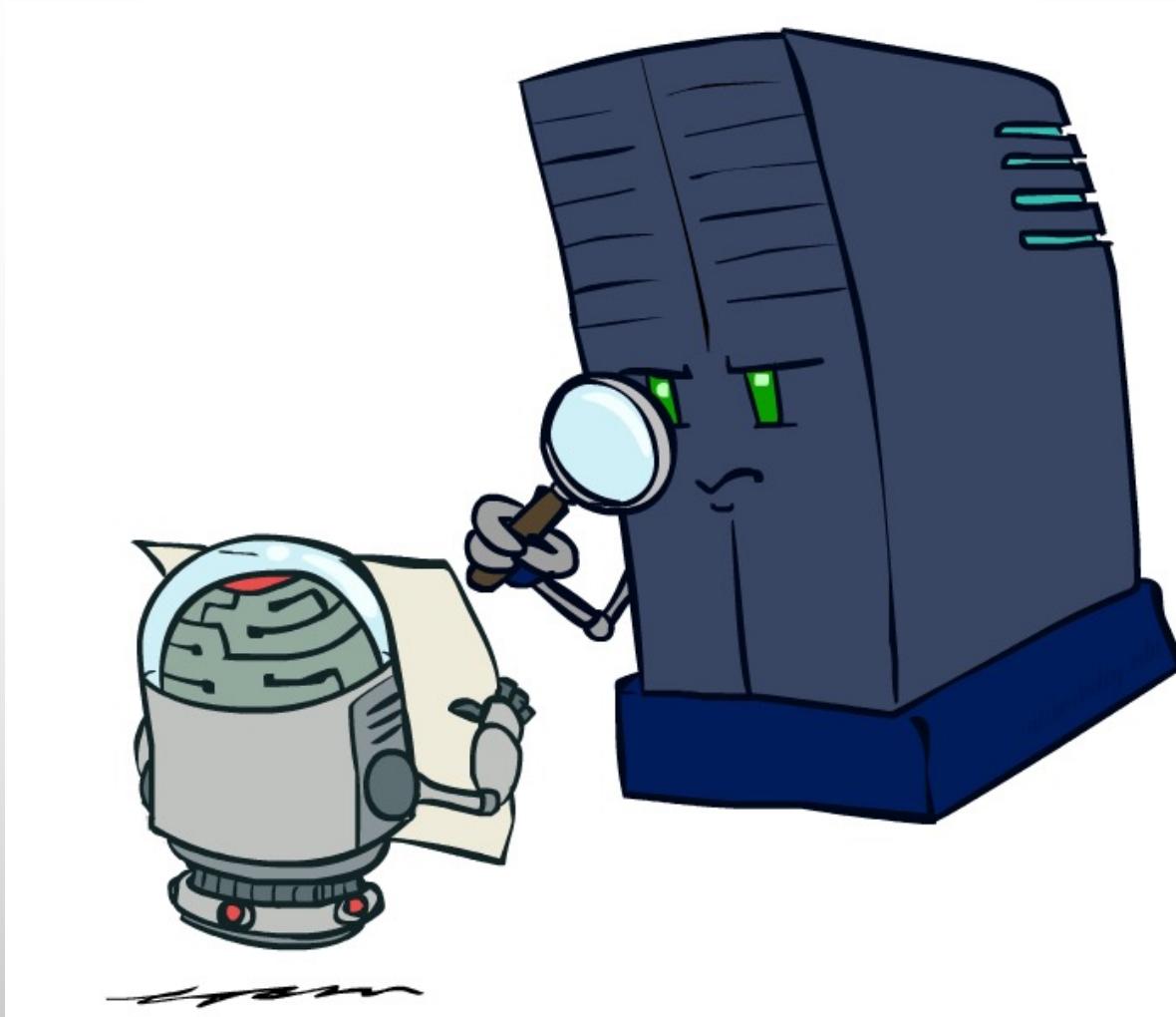
- Value iteration is just a fixed point solution method
 - ... though the V_k vectors are also interpretable as time-limited values



Policy Methods



Policy Evaluation



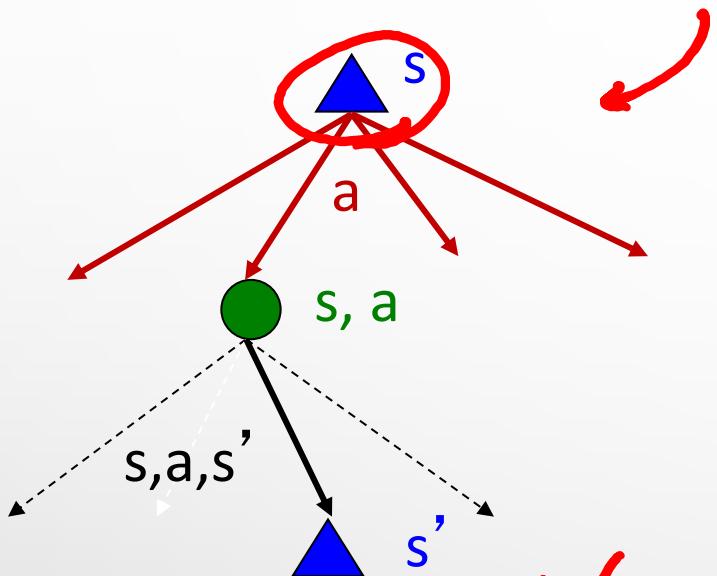
Fixed Policies

$$V_\pi(s) = Q(s, \pi(s))$$

$$Q(s, a) = \sum_{s'} T(s, a, s')$$

Do what π says to do

Do the optimal action



$$V_\pi(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_\pi(s')]$$

- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state
 - ... though the tree's value would depend on which policy we fixed

Utilities for a Fixed Policy

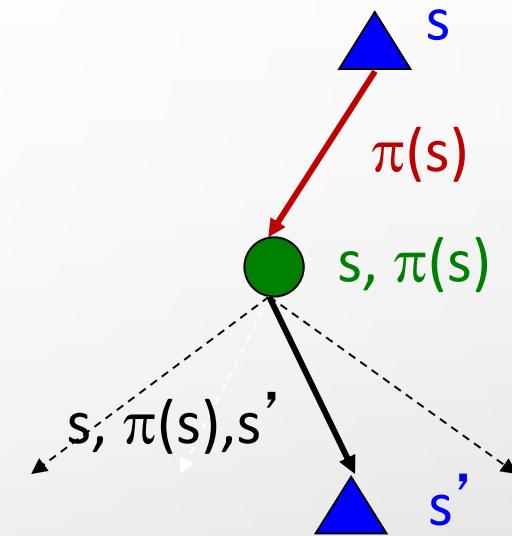
- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- Define the utility of a state s , under a fixed policy π :

$V^\pi(s) = \text{expected total discounted rewards starting in } s \text{ and following } \pi$

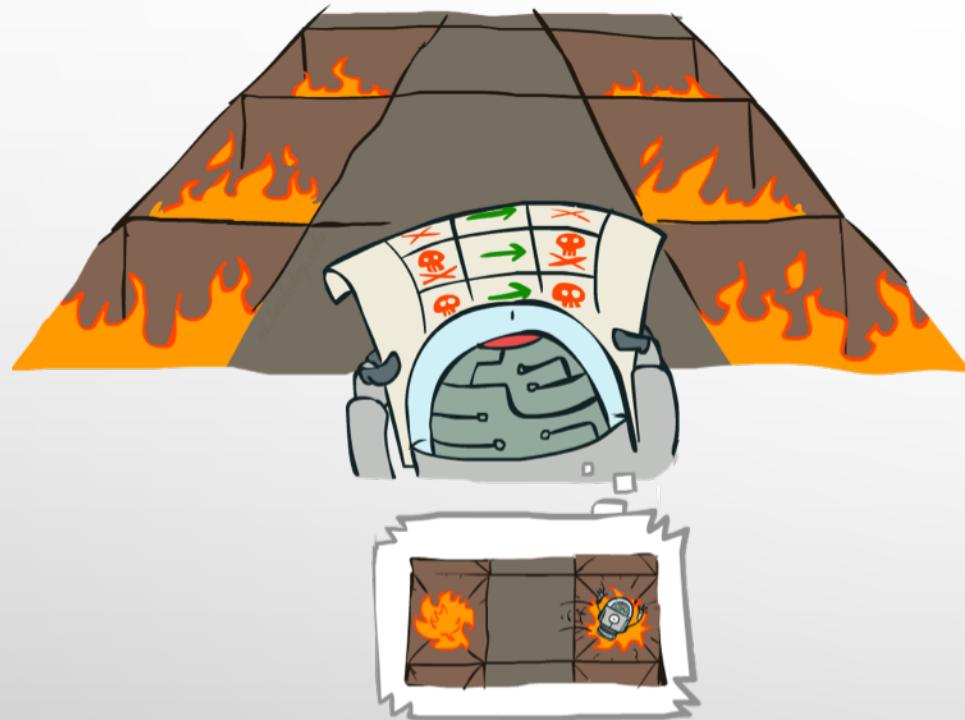
- Recursive relation (one-step look-ahead / bellman equation):

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

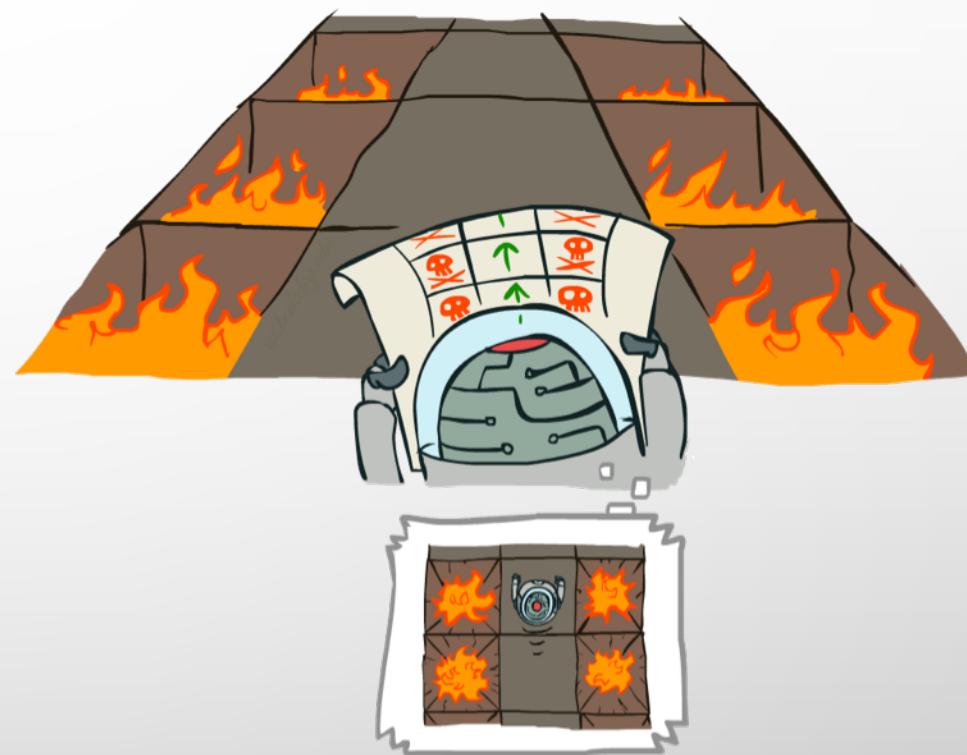


Example: Policy Evaluation

Always Go Right



Always Go Forward



Example: Policy Evaluation

Always Go Right



Always Go Forward



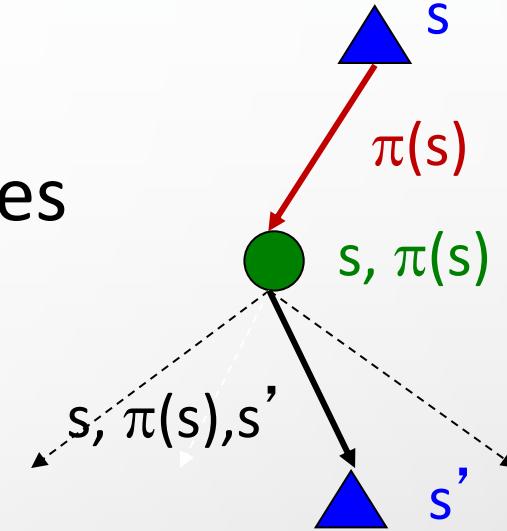
Policy Evaluation

- How do we calculate the V 's for a fixed policy π ?
- Idea 1: turn recursive bellman equations into updates
(Like value iteration)

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

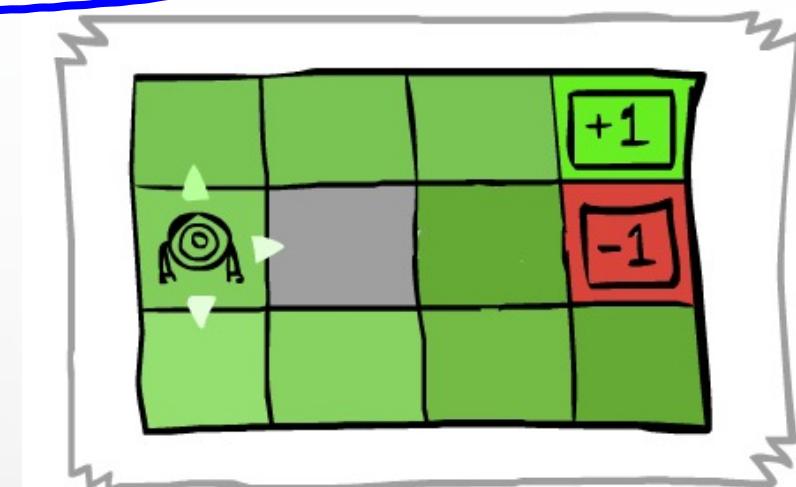
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

- Efficiency: $O(S^2)$ per iteration
- Idea 2: without the maxes, the bellman equations are just a linear system
 - Solve with MATLAB (or your favorite linear system solver)

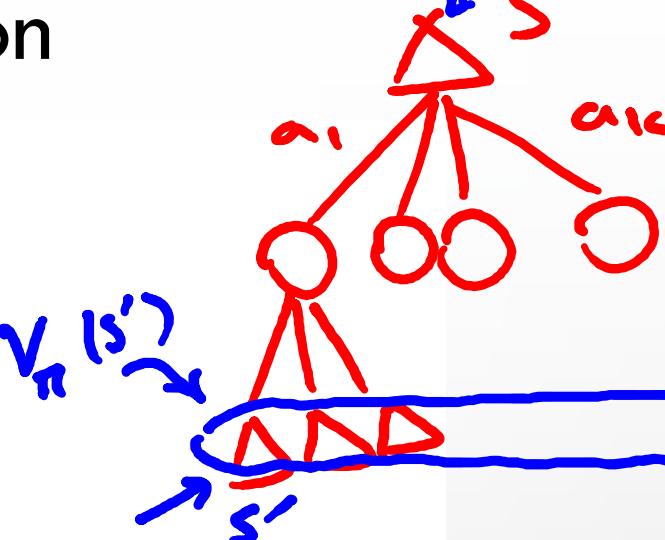


π_{new} \Rightarrow π_{old}

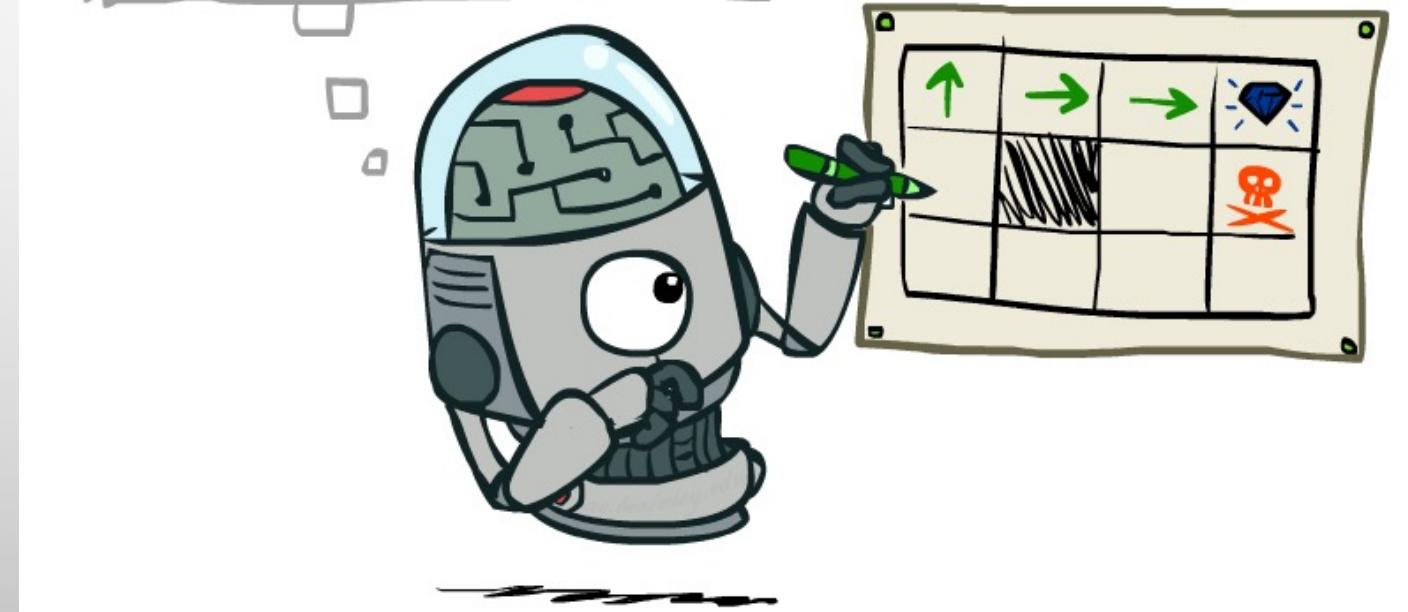
Policy Extraction



$$\pi(s) = \arg\max_{a_j} Q(s, a_j)$$

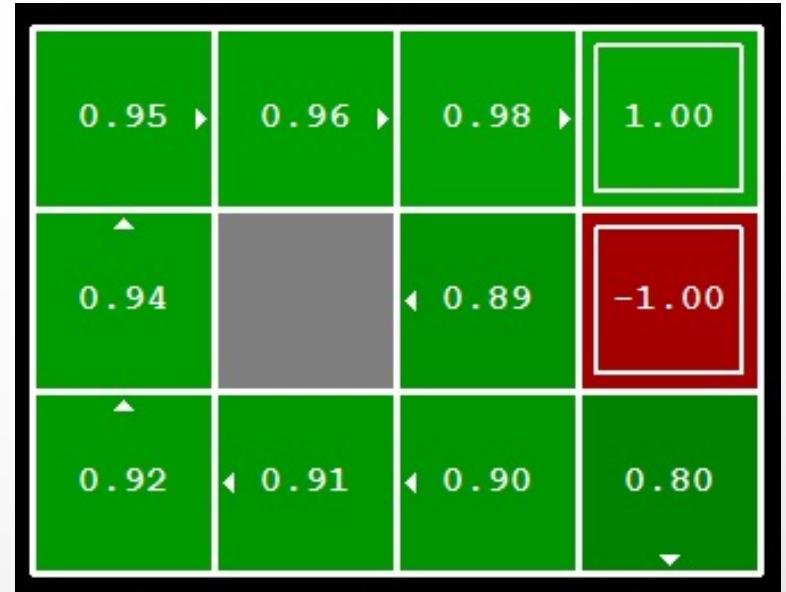


depth
2
exp



Computing Actions from Values

- Let's imagine we have the optimal values $V^*(s)$
- How should we act?
 - It's not obvious!
- We need to do a mini-expectimax (one step)



$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

- This is called **policy extraction**, since it gets the policy implied by the values

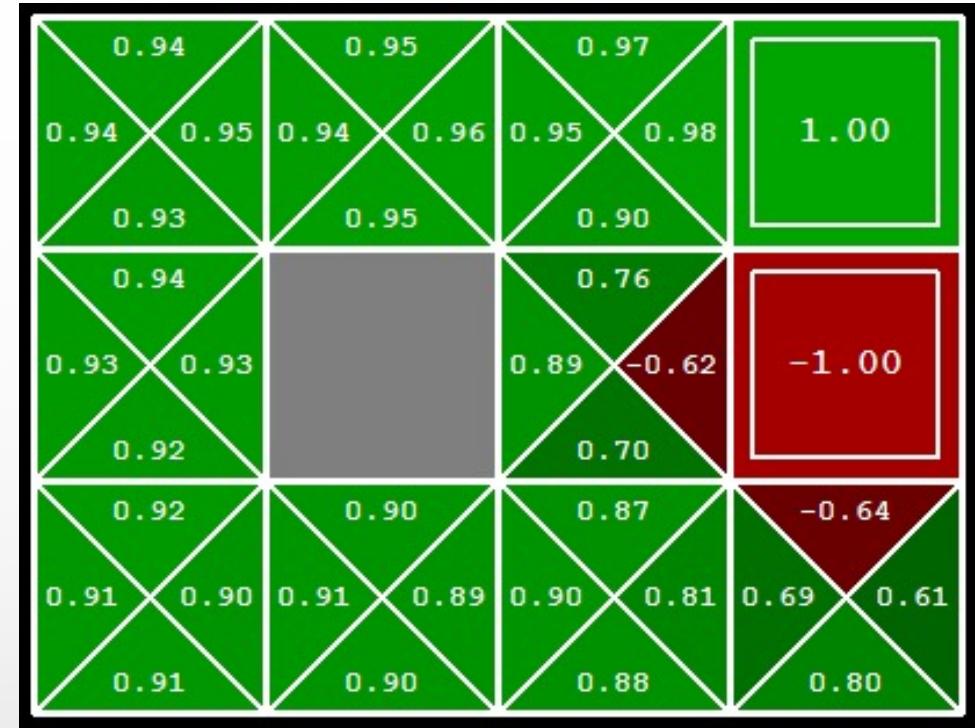
Computing Actions from Q-Values

- Let's imagine we have the optimal Q-values:

- How should we act?

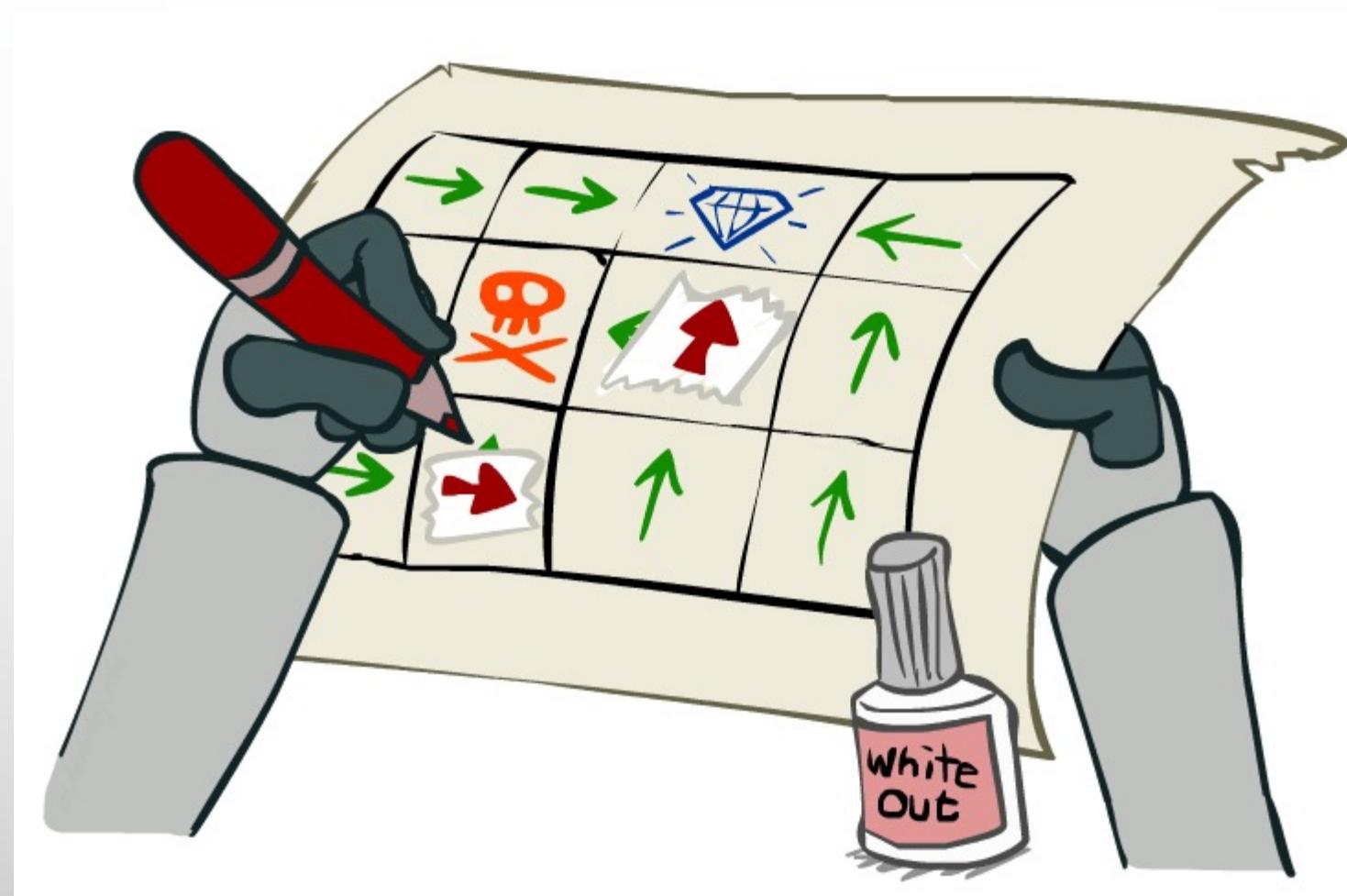
- Completely trivial to decide!

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



- Important lesson: actions are easier to select from q-values than values!

Policy Iteration

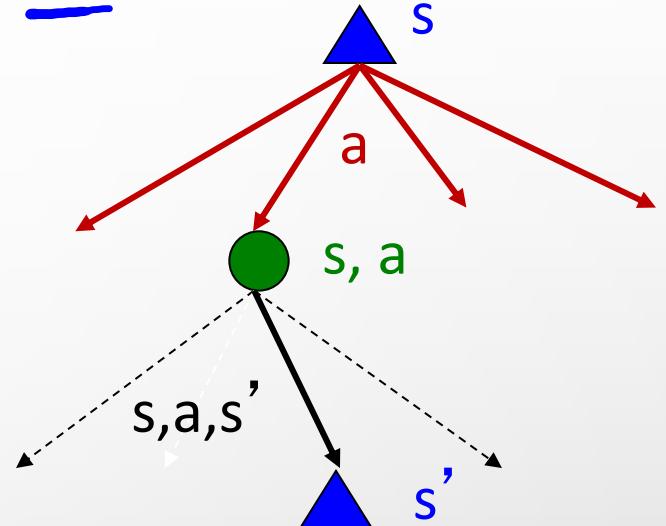


Problems with Value Iteration

Reinf. Learning

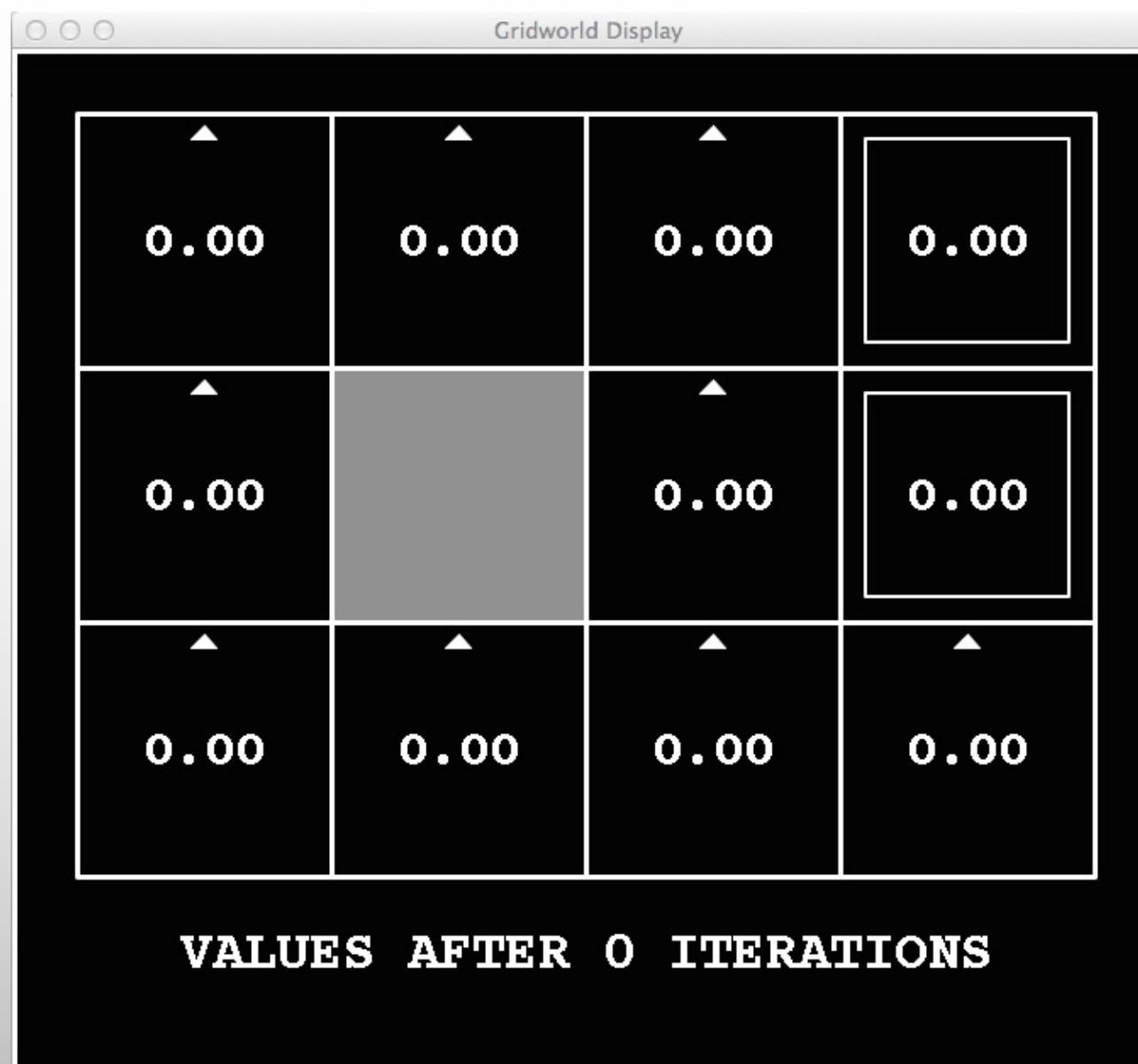
- Value iteration repeats the Bellman updates:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

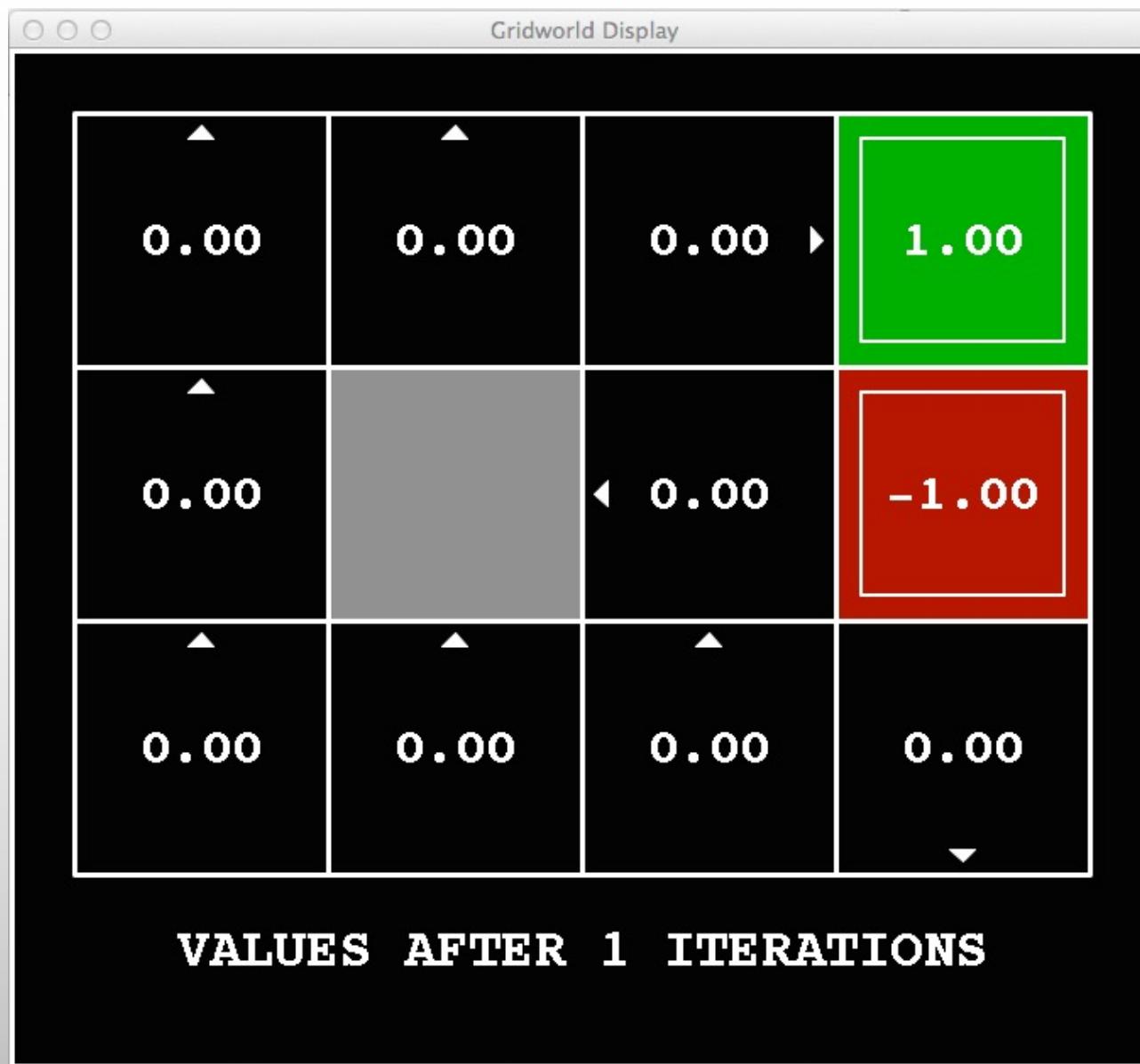


- Problem 1: it's slow – $O(S^2A)$ per iteration
- Problem 2: the “max” at each state rarely changes
- Problem 3: the policy often converges long before the values

$k=0$



$k=1$



k=2



k=3



k=4



k=5



VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



VALUES AFTER 6 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

$k=7$



k=8



k=9



k=10



$k=11$



k=12



k=100



Policy Iteration

- Alternative approach for optimal values:
 - **Step 1: policy evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence
 - **Step 2: policy improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
 - Repeat steps until policy converges
- This is **policy iteration**
 - It's still optimal!
 - Can converge (much) faster under some conditions

Policy Iteration

- Evaluation: for fixed current policy π , find values with policy evaluation:
 - Iterate until values converge:

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$$

- Improvement: for fixed values, get a better policy using policy extraction
 - One-step look-ahead:

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s') \right]$$

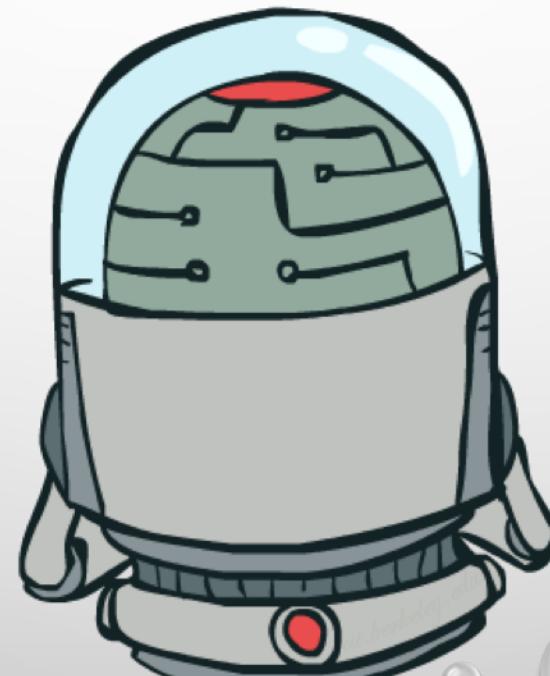
Comparison

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
 - Every iteration updates both the values and (implicitly) the policy
 - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
 - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
 - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
 - The new policy will be better (or we're done)
- Both are dynamic programs for solving MDPs

Summary: MDP Algorithms

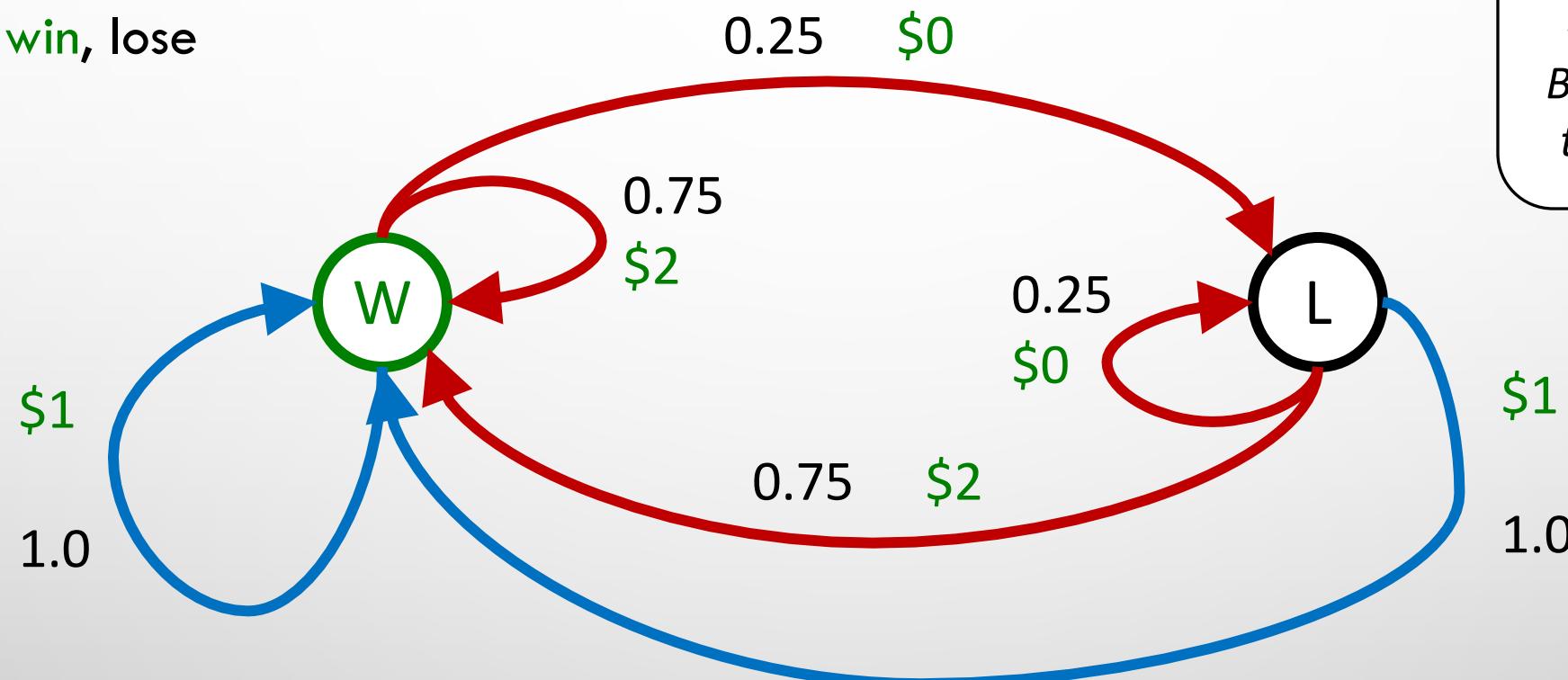
- So you want to....
 - Compute optimal values: use value iteration or policy iteration
 - Compute values for a particular policy: use policy evaluation
 - Turn your values into a policy: use policy extraction (one-step lookahead)
- These all look the same!
 - They basically are – they are all variations of bellman updates
 - They all use one-step lookahead expectimax fragments
 - They differ only in whether we plug in a fixed policy or max over actions

Double Bandits



Double-Bandit MDP

- Actions: *blue*, *red*
- States: *win*, *lose*



No discount
100 time steps
Both states have the same value

Offline Planning

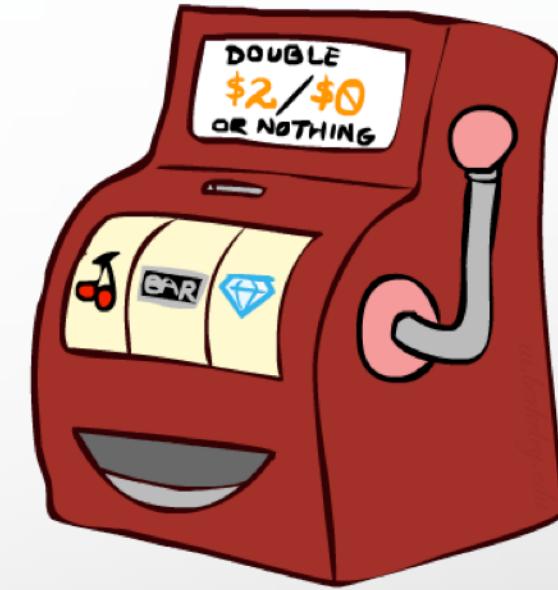
- Solving MDPs is offline planning
 - You determine all quantities through computation
 - You need to know the details of the MDP
 - You do not actually play the game!

*No discount
100 time steps
Both states have
the same value*

	Value
Play Red	150
Play Blue	100



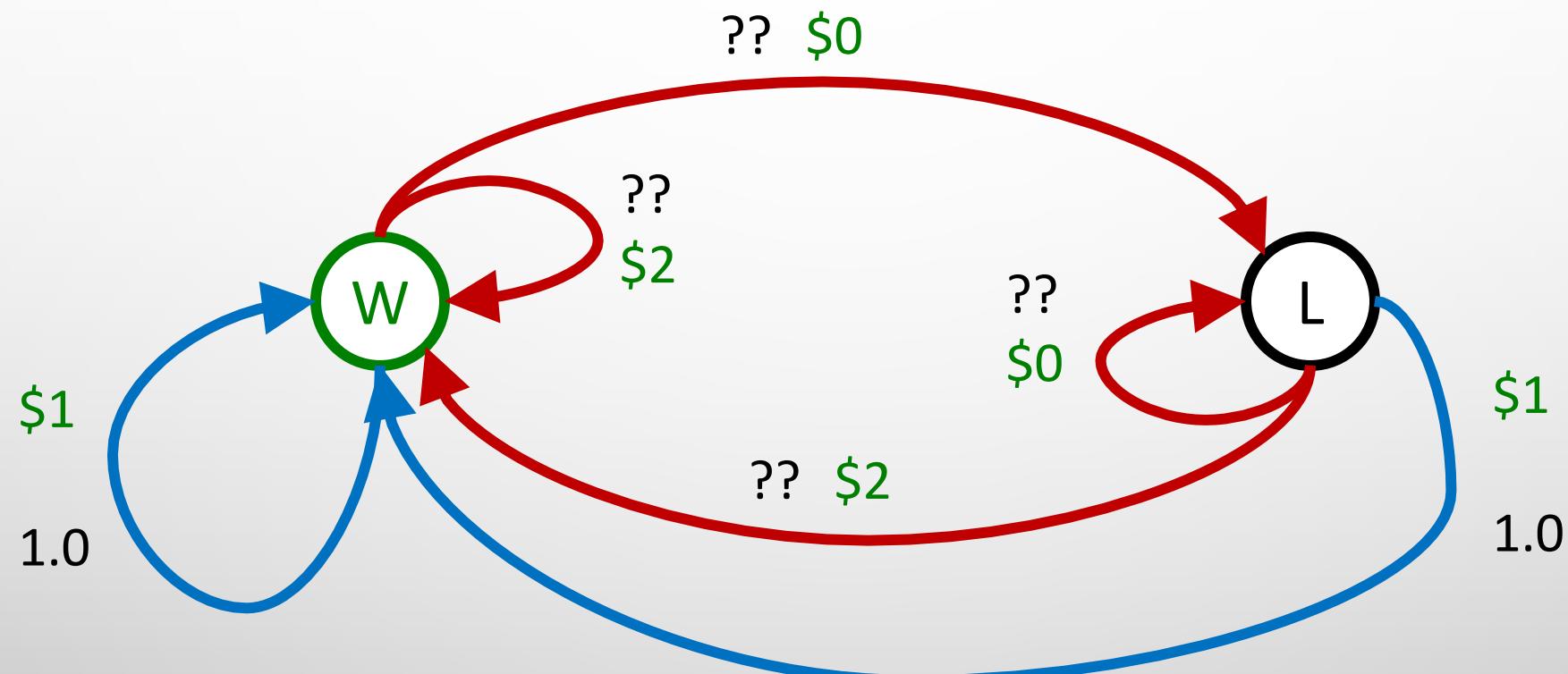
Let's Play!



\$2	\$2	\$0	\$2	\$2
\$2	\$2	\$0	\$0	\$0

Online Planning

- Rules changed! Red's win chance is different.



Let's Play!



\$0 \$0 \$0 \$2 \$0
\$2 \$0 \$0 \$0 \$0

What Just Happened?

- That wasn't planning, it was learning!
 - Specifically, reinforcement learning
 - There was an MDP, but you couldn't solve it with just computation
 - You needed to actually act to figure it out
- Important ideas in reinforcement learning that came up
 - Exploration: you have to try unknown actions to get information
 - Exploitation: eventually, you have to use what you know
 - Regret: even if you learn intelligently, you make mistakes
 - Sampling: because of chance, you have to try things repeatedly
 - Difficulty: learning can be much harder than solving a known MDP



Next Time: Reinforcement Learning!