

# Announcements

---

- **Homework 3 due today (Sept 24) at 11:59pm PT**
- **Project 2 due next Wednesday (Oct 2) at 11:59pm PT**

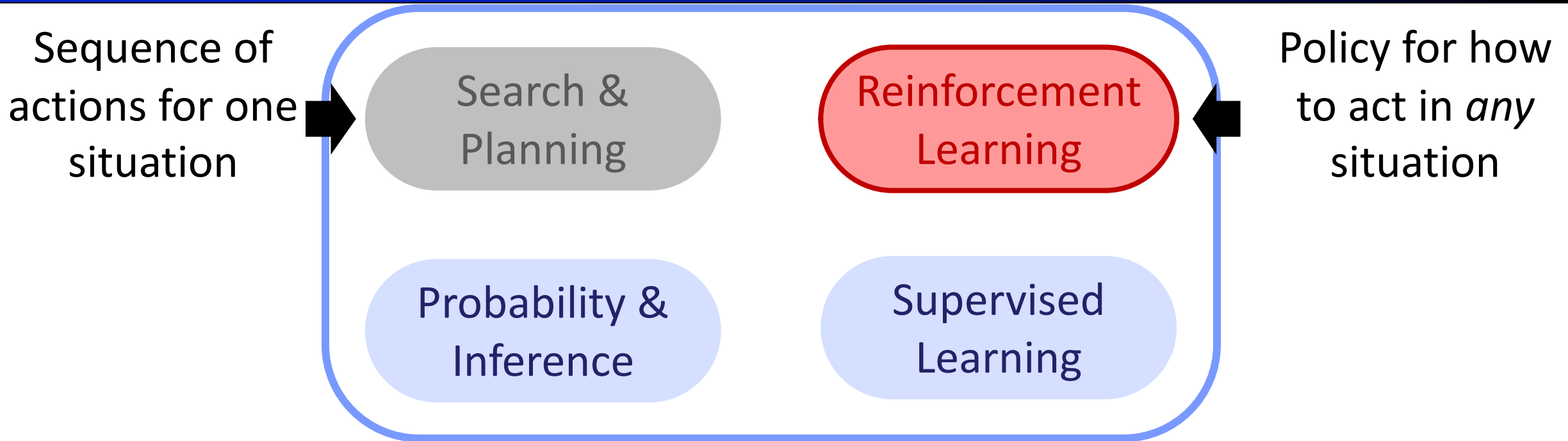
# CS 188: Artificial Intelligence

## Markov Decision Processes I



University of California, Berkeley

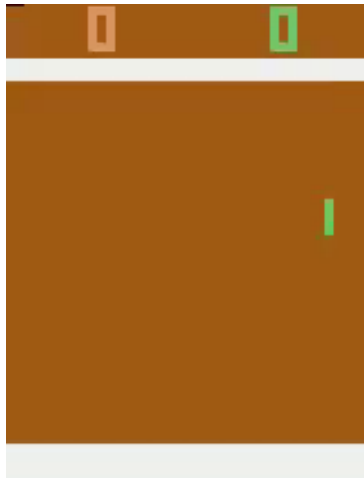
# Preview of Next Two Weeks



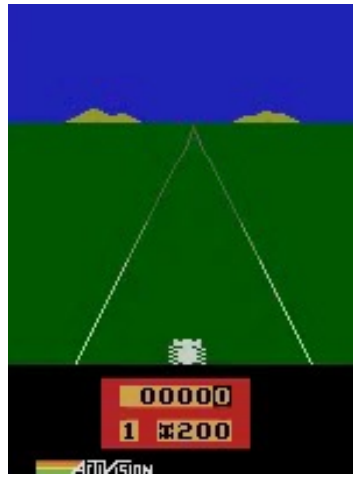
- This week: pre-compute policies
  - Know the model of the world
- Next week: learn policies from trial and error
  - Learn only from interactions with the world

# Examples of (Deep) Reinforcement Learning

2013: Playing Atari games



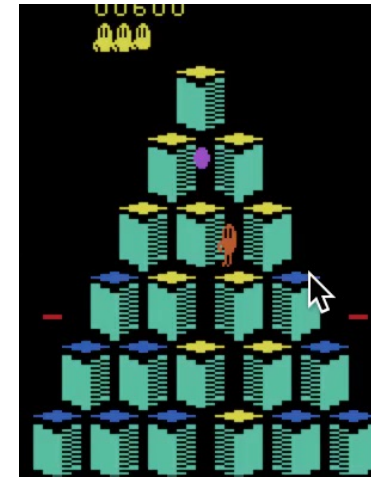
Pong



Enduro



Beamrider

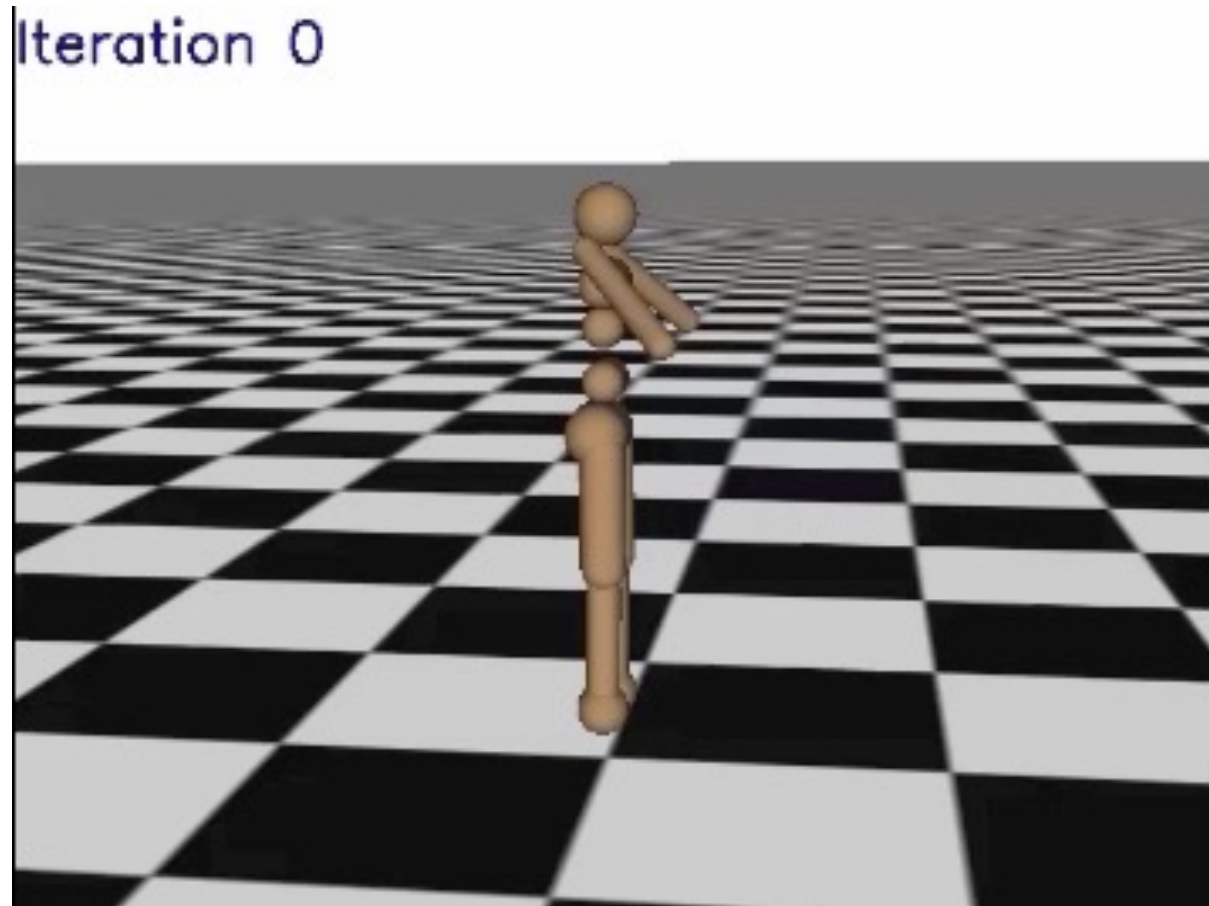


Q\*bert

[Human-level control through deep reinforcement learning. Mnih et al. Nature 2015]

# Examples of (Deep) Reinforcement Learning

2015: Locomotion from trial and error



[Trust Region Policy Optimization. Schulman et al. ICLR 2015]

# Examples of (Deep) Reinforcement Learning

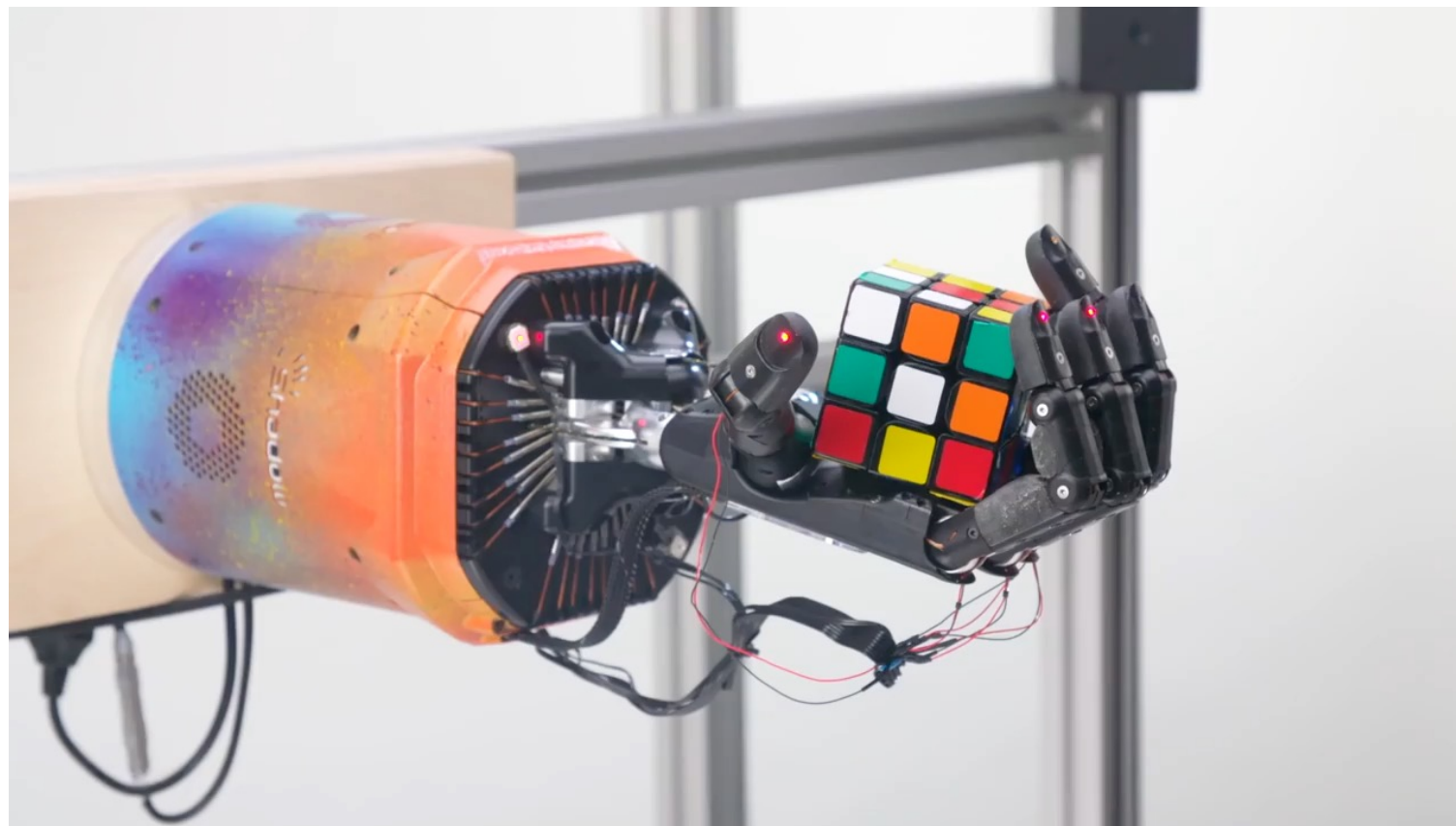
2016: Playing Go (and beating human champion)



[Mastering the game of Go with deep neural networks and tree search. Silver et al. Nature 2016]

# Examples of (Deep) Reinforcement Learning

2019: Robot manipulation



[Solving Rubik's cube with a robot hand. OpenAI. 2019]



# Examples of (Deep) Reinforcement Learning

## 2022: Nuclear fusion plasma control

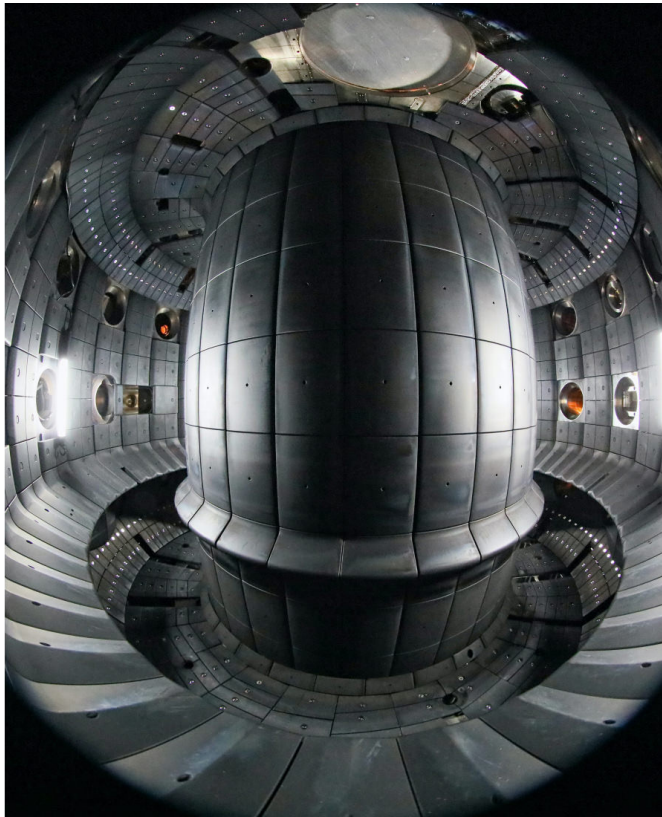
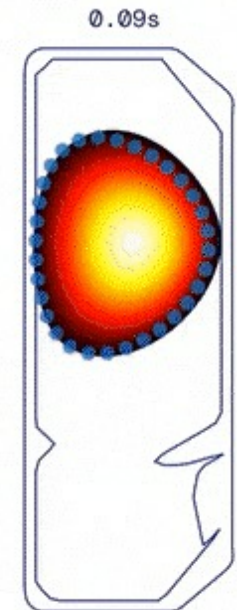


Photo Credits: DeepMind and SPC/EPFL



View from inside the tokamak



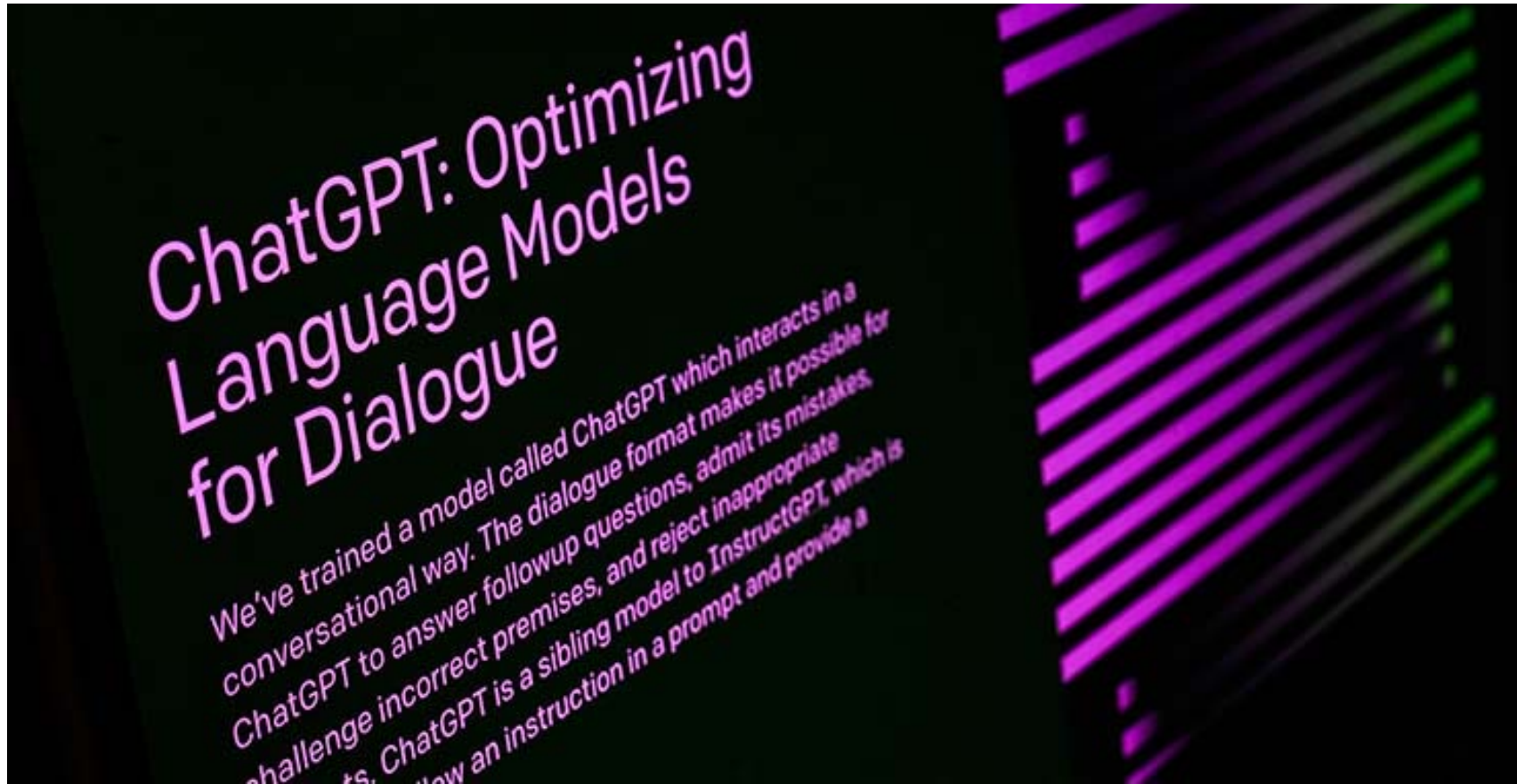
Plasma state reconstruction

[Magnetic control of tokamak plasmas through deep reinforcement learning. Degraeve et al. Nature 2022]



# Examples of (Deep) Reinforcement Learning

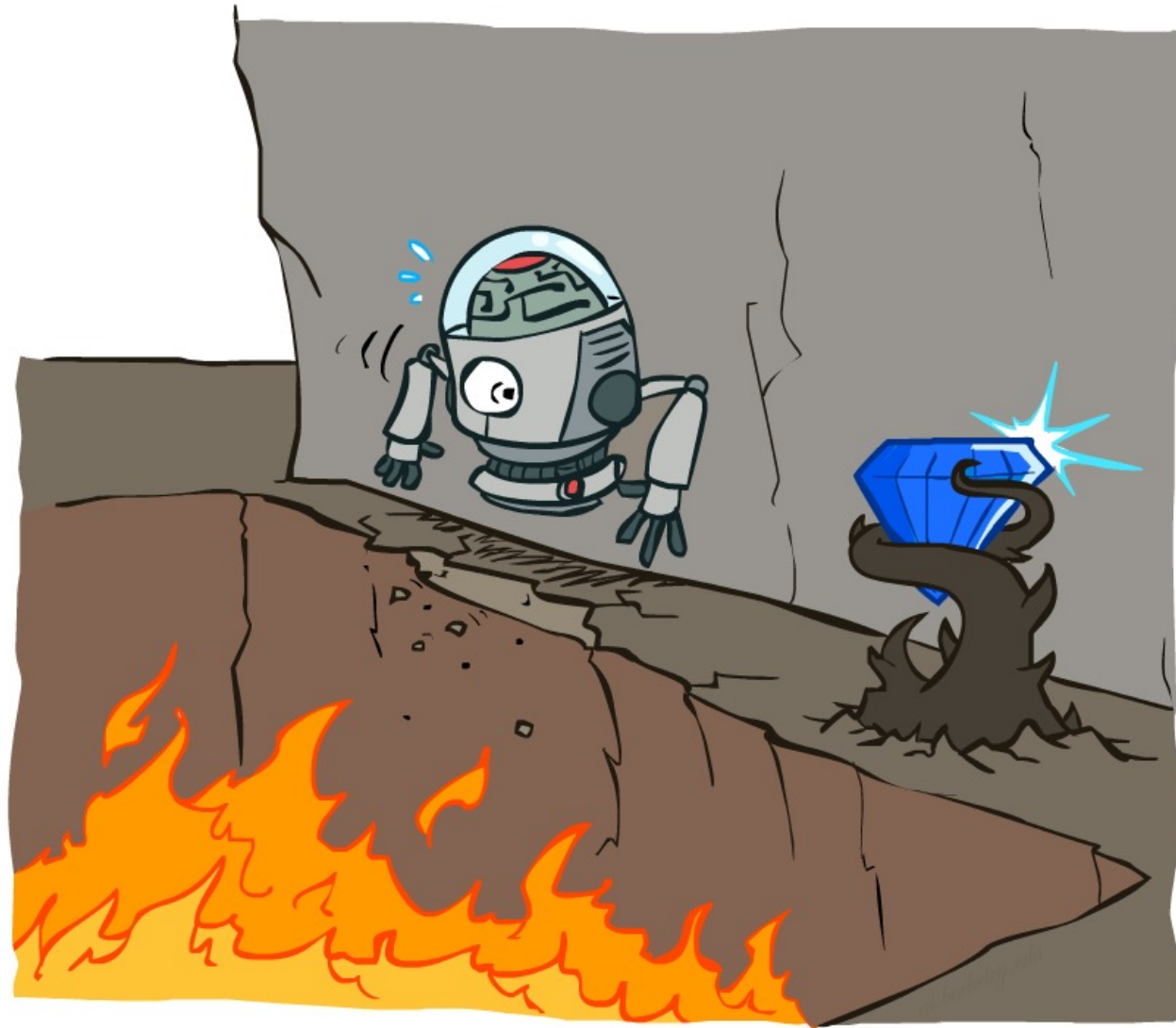
## 2022: Training Language Assistants with Human Feedback



[Aligning language models to follow instructions. Ouyang et al. 2022]

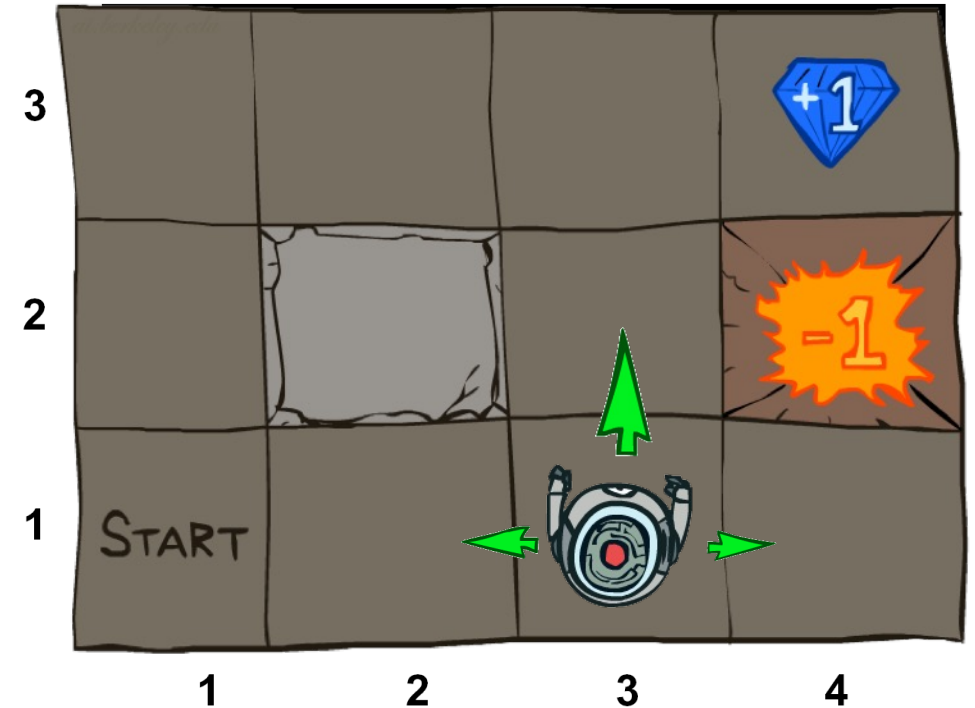
# Non-Deterministic Search

---



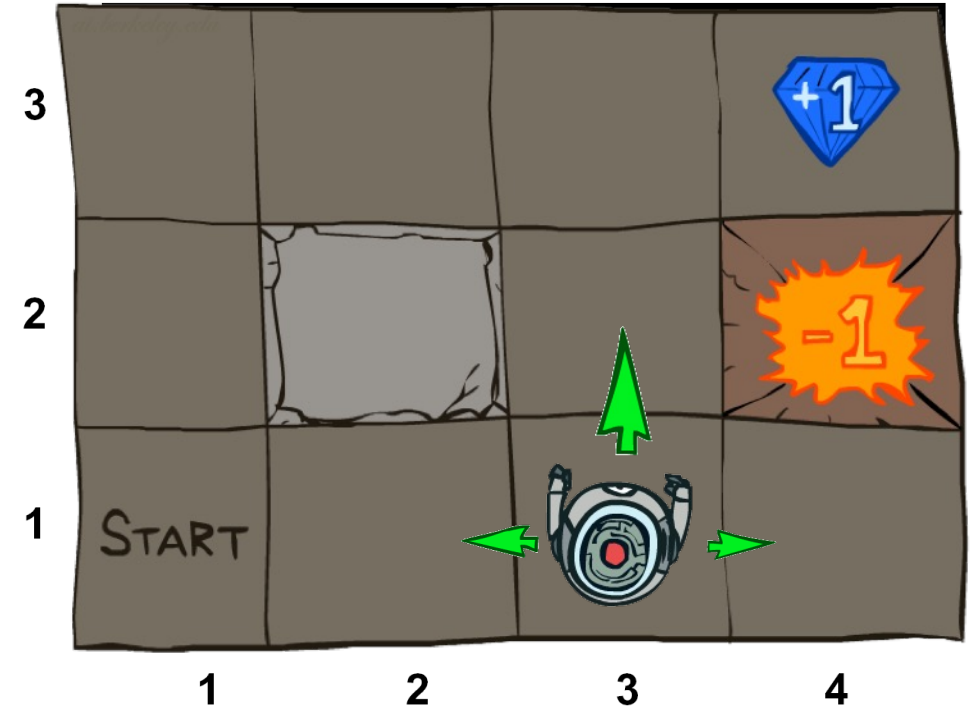
# Example: Grid World

- 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: maximize sum of rewards

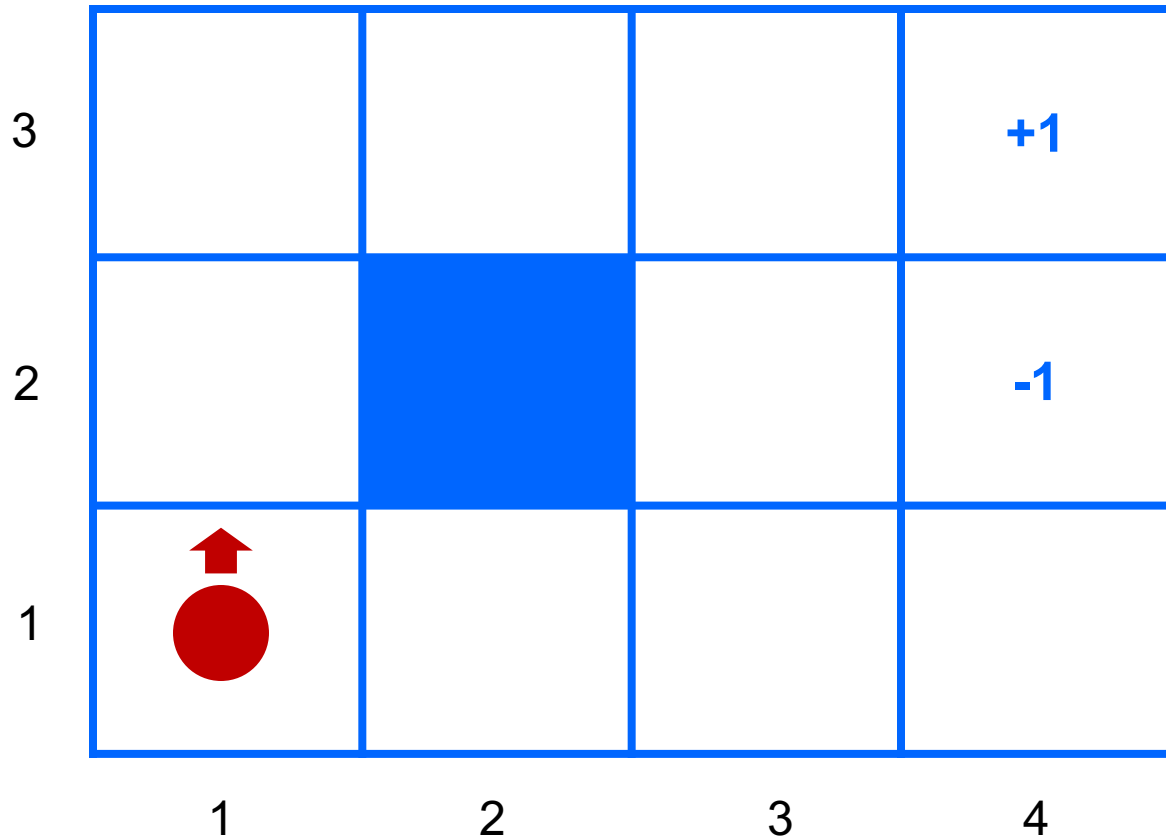


# Markov Decision Processes

- 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**



# Gridworld Example

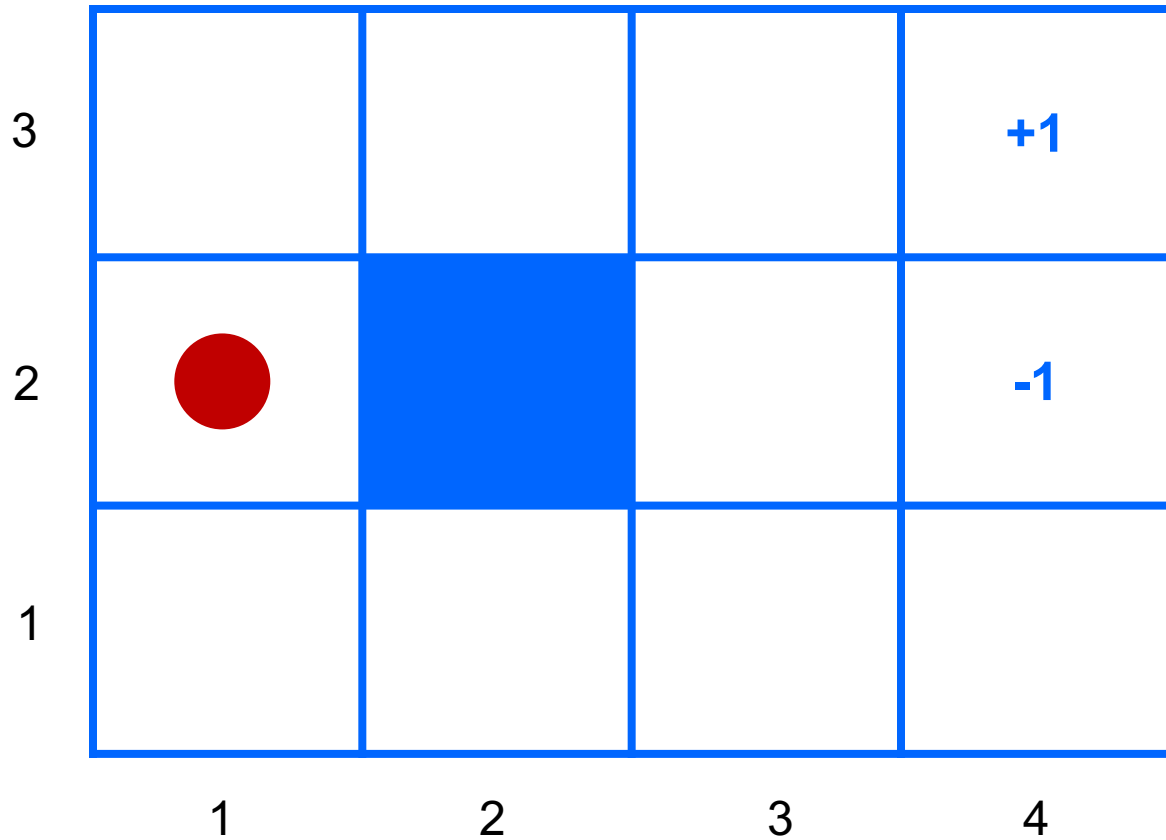


s	a	s'	R
(1,1)	north		

**T(s, a, s')**:

- $T((1,1), \text{north}, (2,1)) = 0.8$
- $T((1,1), \text{north}, (1,2)) = 0.1$
- $T((1,1), \text{north}, (1,1)) = 0.1$

# Gridworld Example



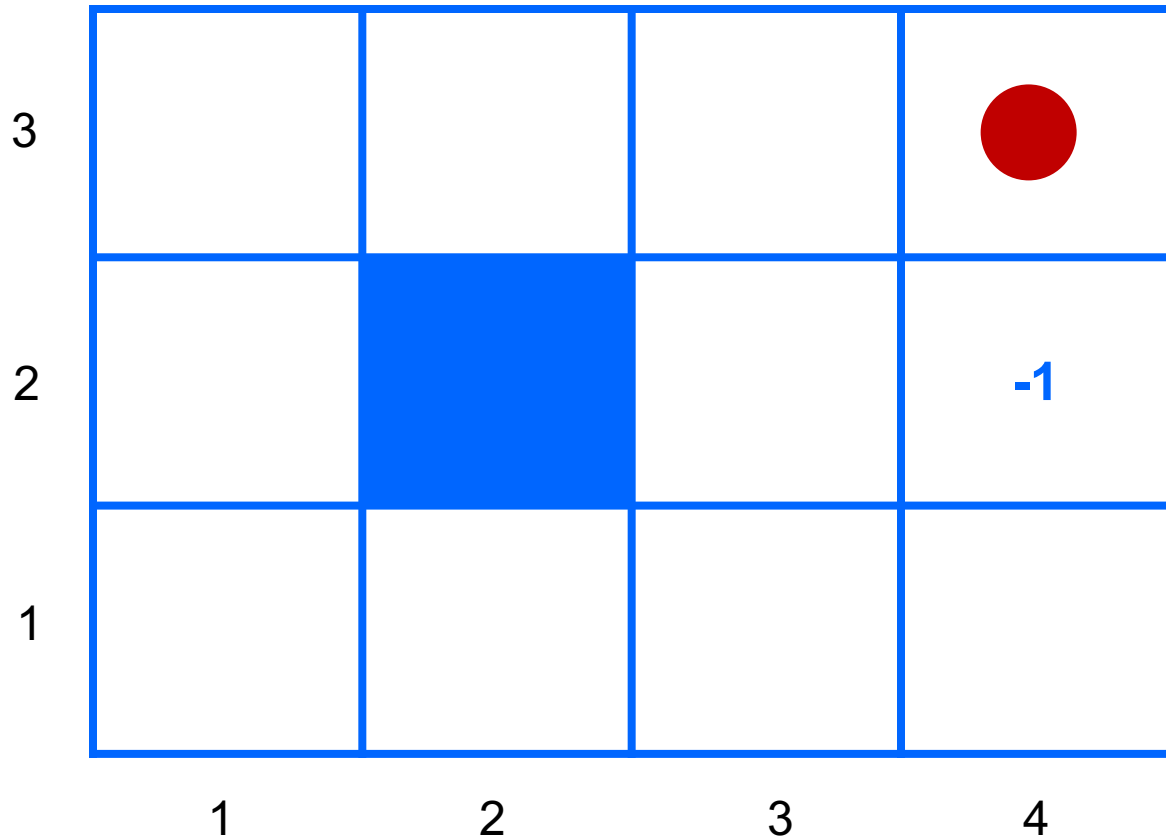
s	a	s'	R
(1,1)	north	(2,1)	-0.1

**$R(s, a, s')$ :**

$$R((1,1), \text{north}, (2,1)) = -0.1$$



# Gridworld Example



<b>s</b>	<b>a</b>	<b>s'</b>	<b>R</b>
(1,1)	north	(2,1)	-0.1
(2,1)	north	(3,1)	-0.1
(3,1)	east	(2,1)	-0.1
(2,1)	north	(3,1)	-0.1
(3,1)	east	(2,3)	-0.1
(2,3)	east	(3,3)	-0.1
(3,3)	east	(4,3)	-0.1
(4,3)	exit	(4,3)	+1

# 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

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

=

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

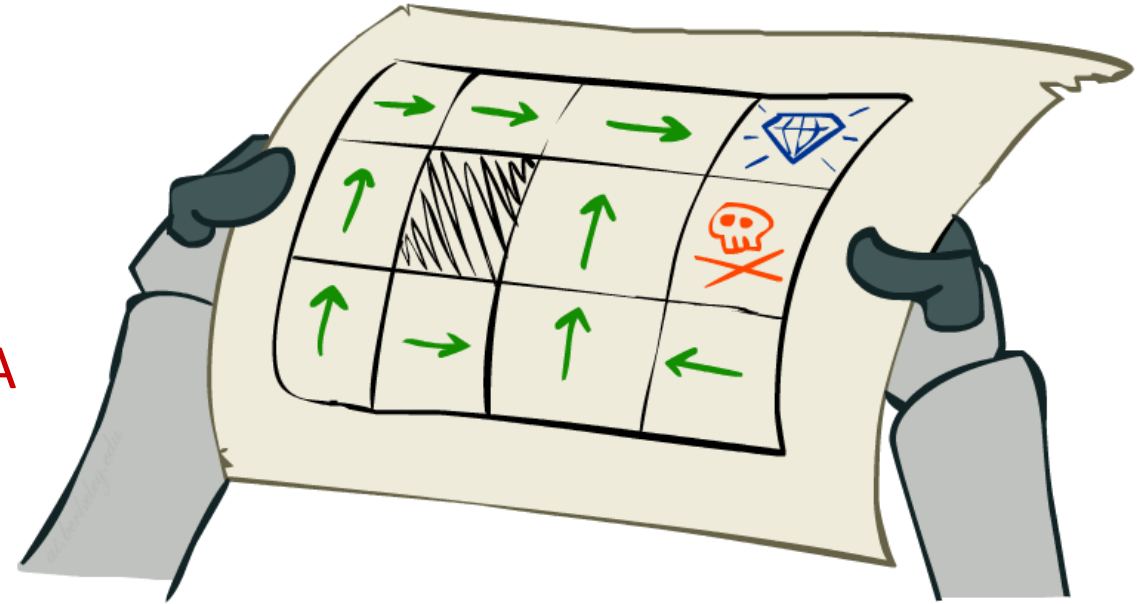
- This is just like search, where the successor function could only depend on the current state (not the history)



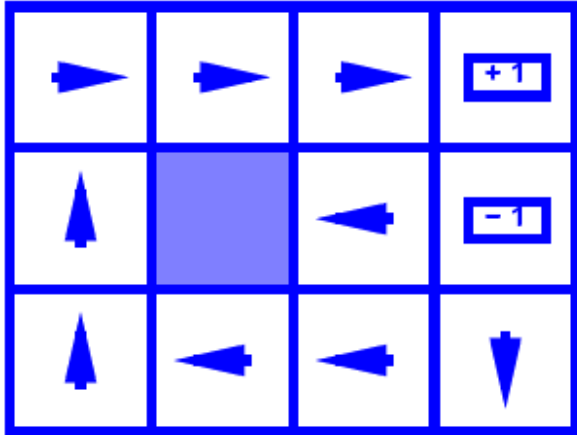
Andrey Markov  
(1856-1922)

# Policies

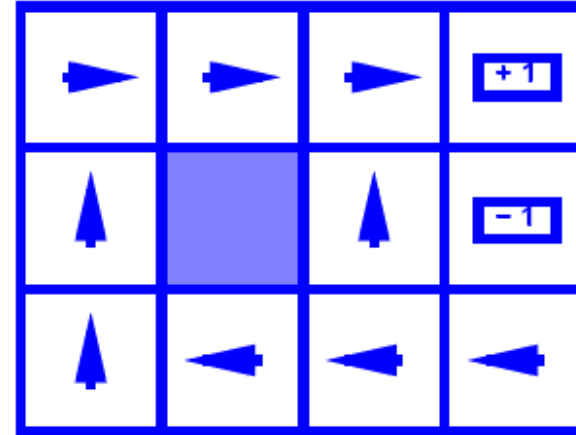
- In deterministic single-agent search problems, we wanted an optimal **plan**, or sequence of actions, from start to a goal
- For MDPs, we want an optimal **policy**  $\pi^*: S \rightarrow A$ 
  - A policy  $\pi$  gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent



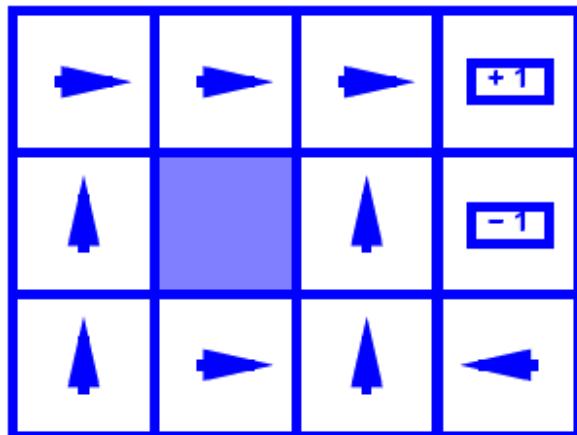
# Optimal Policies



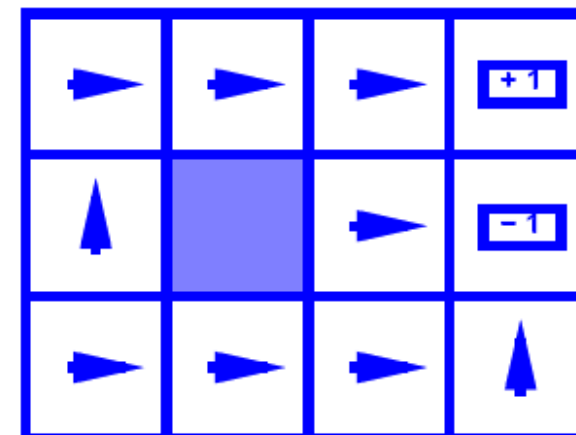
$$R(s) = -0.01$$



$$R(s) = -0.03$$



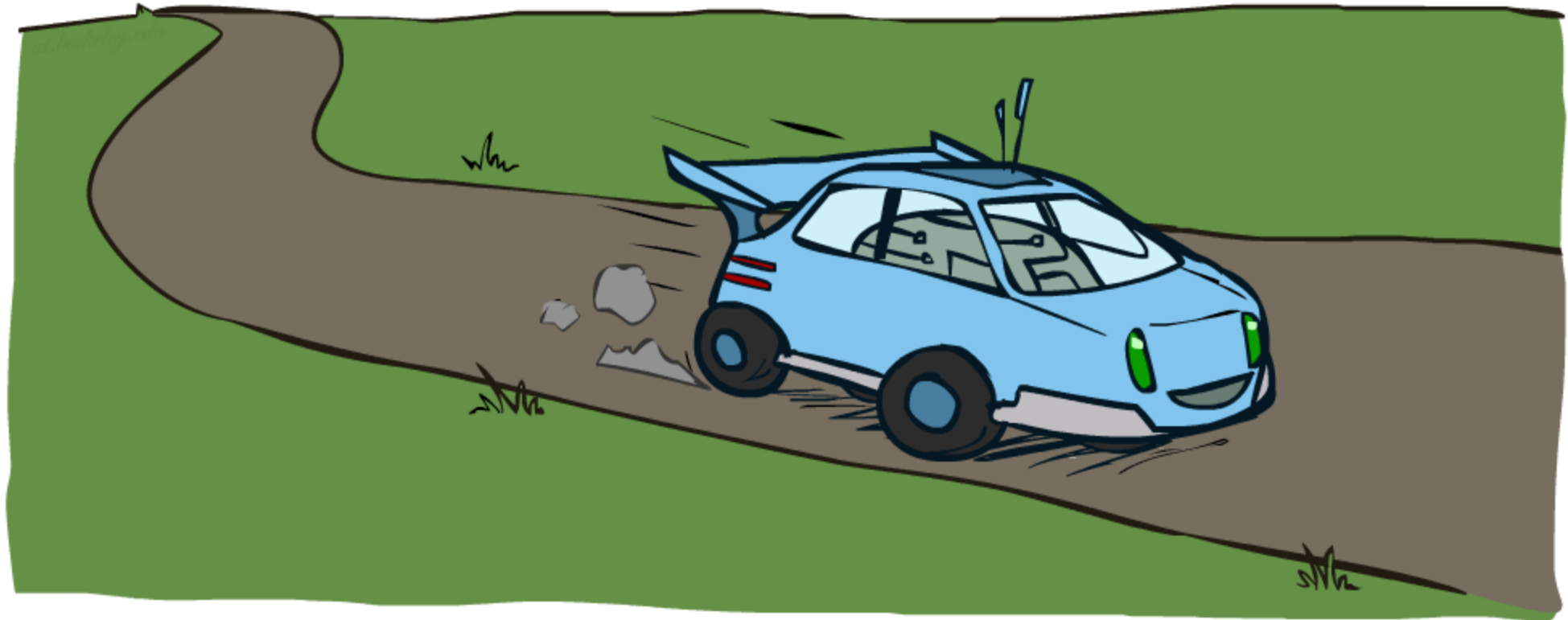
$$R(s) = -0.4$$



$$R(s) = -2.0$$

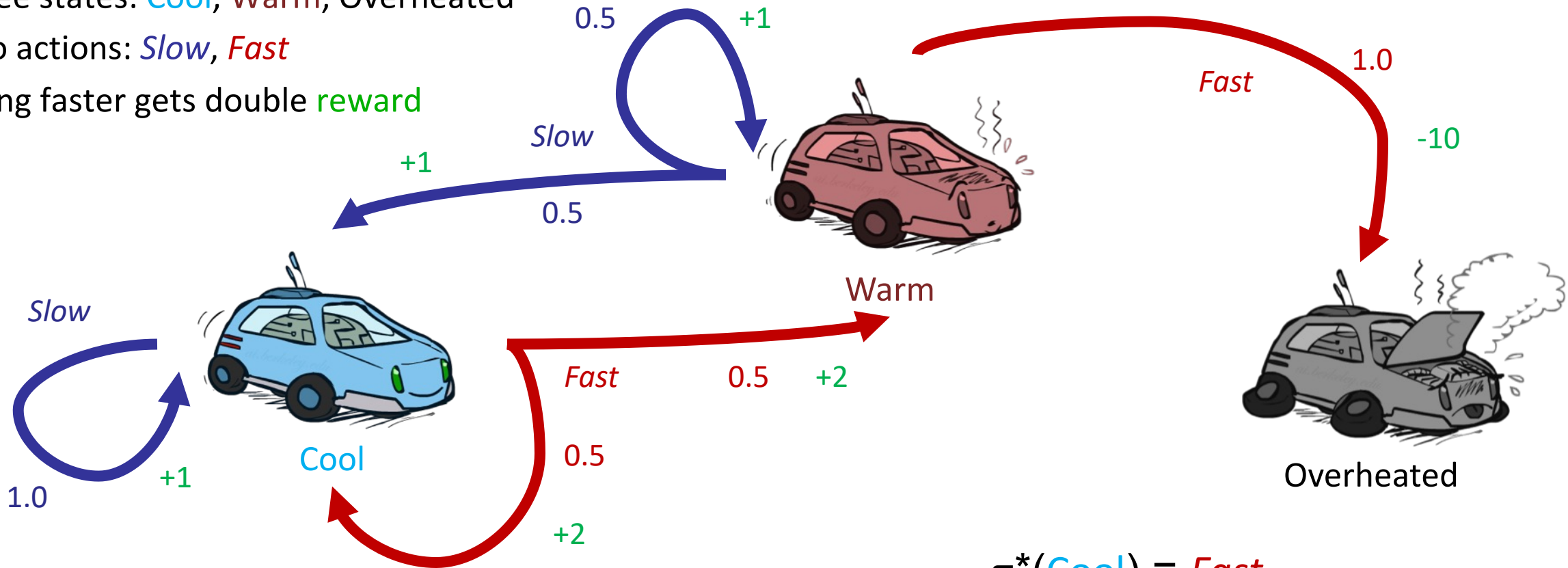
# Example: Racing

---



# Example: Racing

- A robot car wants to travel far, quickly
- Three states: **Cool**, **Warm**, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double **reward**



Optimal Policy:

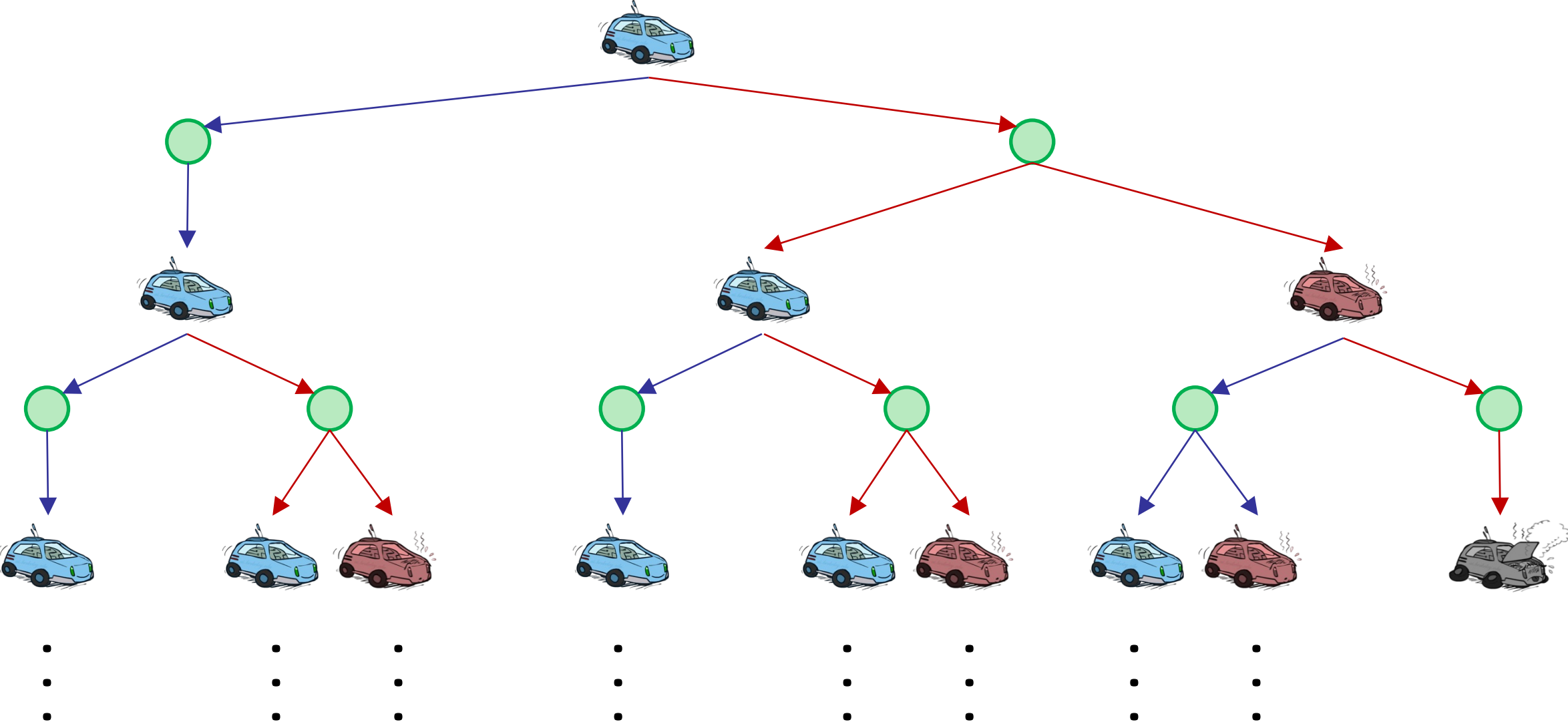
$$\pi^*(\text{Cool}) = \text{Fast}$$

$$\pi^*(\text{Warm}) = \text{Slow}$$

$$\pi^*(\text{Overheated}) = \text{end}$$

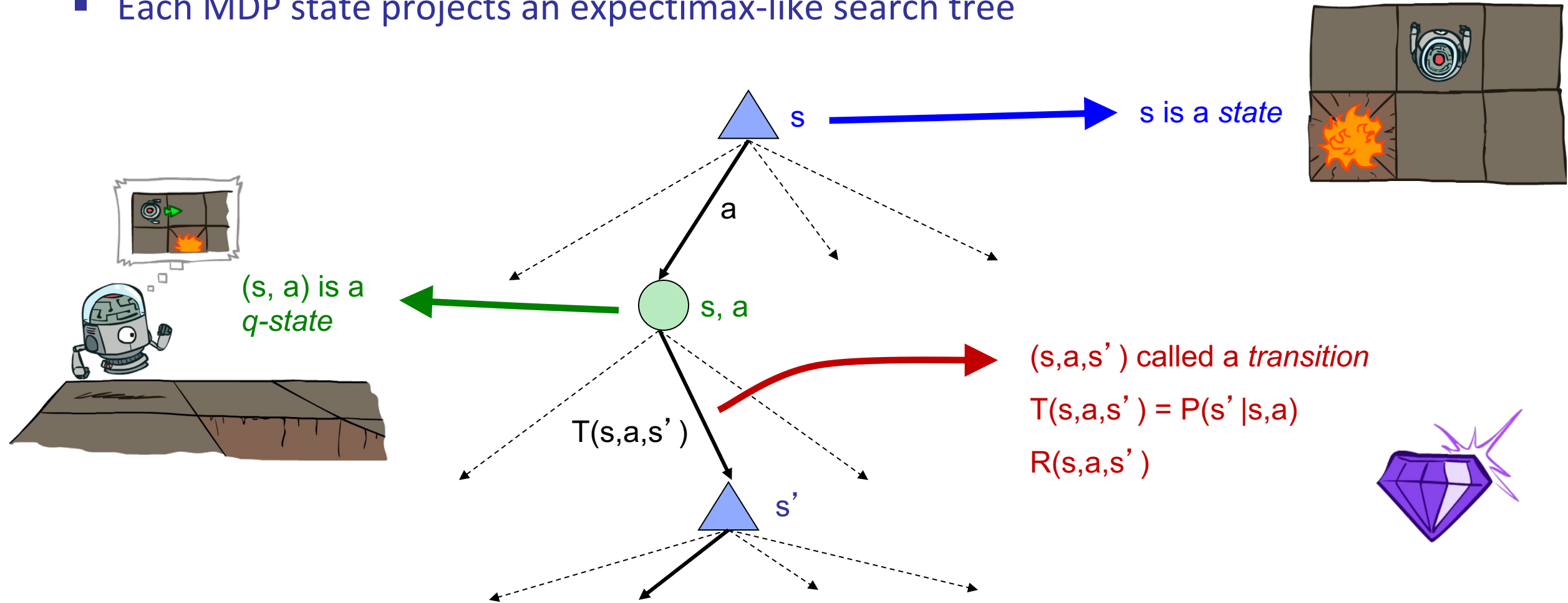


# Racing Search Tree



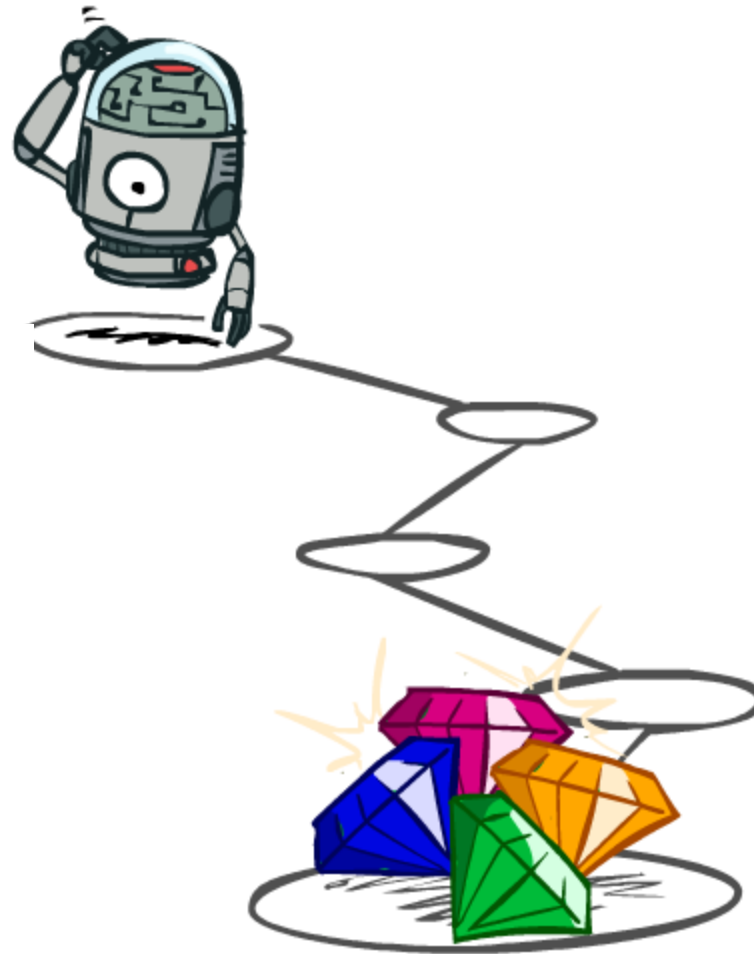
# MDP Search Trees

- Each MDP state projects an expectimax-like search tree



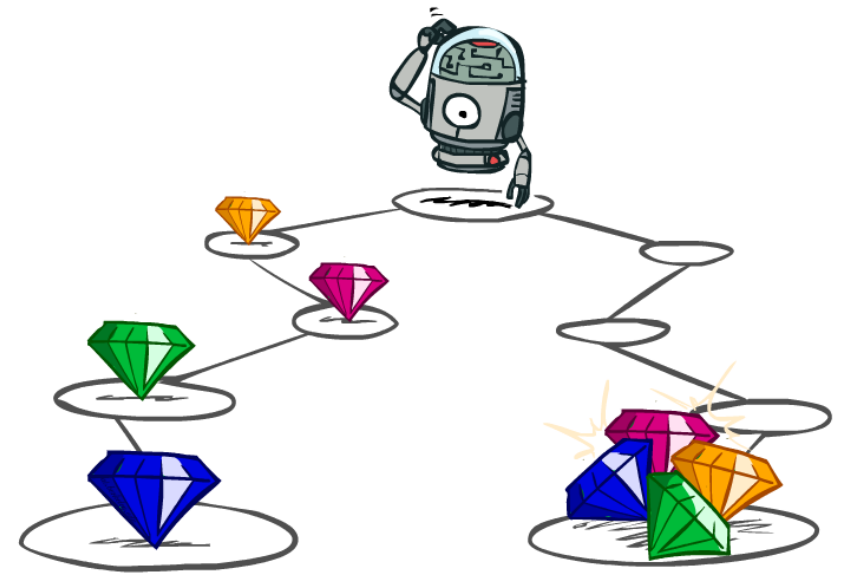
# Utilities of Sequences

---



# Utilities of Sequences

- 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



$\gamma$

Worth Next Step



$\gamma^2$

Worth In Two Steps

( $\gamma$  is between 0 and 1)

# Discounting

- How to discount?

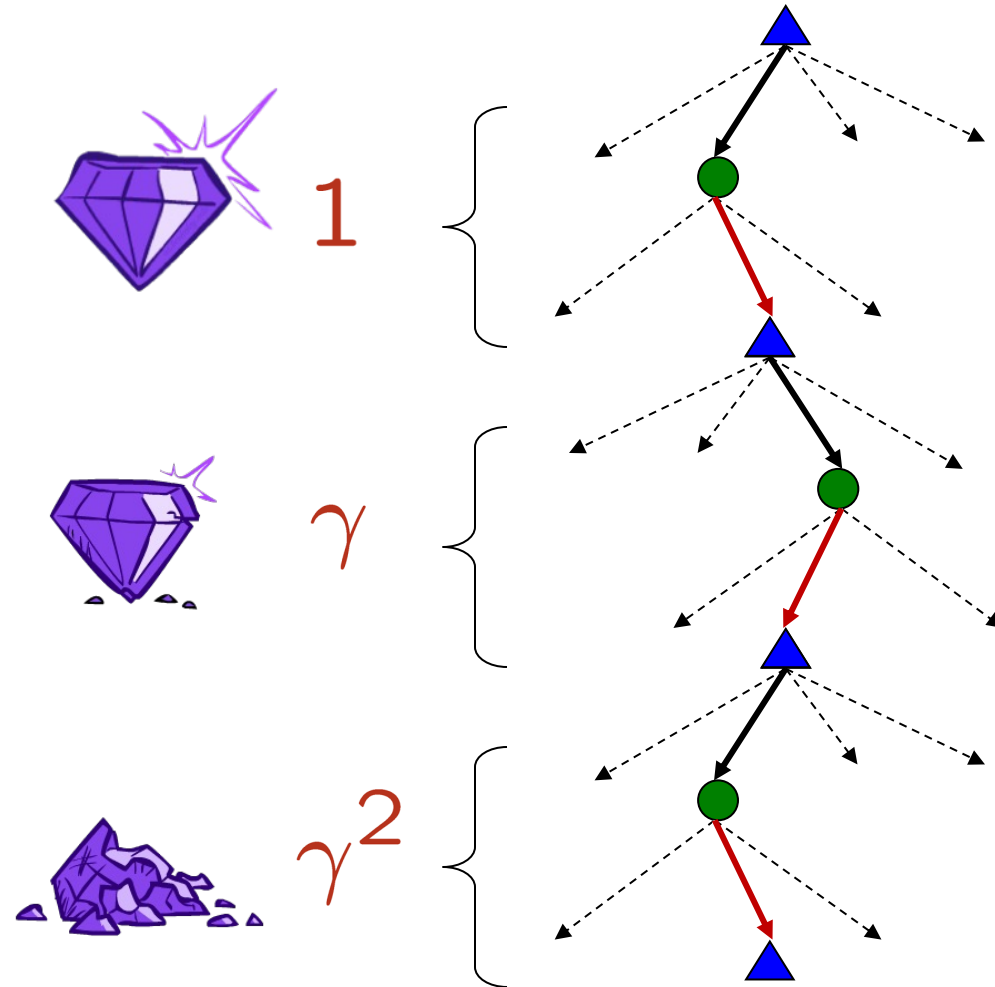
- Each time we descend a level, we multiply in the discount once

- Why discount?

- Sooner rewards probably do have higher utility than later rewards
- Also helps our algorithms converge

- Example: discount of 0.5

- $U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3 = 2.75$
- $U([3,2,1]) = 1*3 + 0.5*2 + 0.25*1 = 5.25$
- $U([1,2,3]) < U([3,2,1])$





# Discounting in Public Policy

COUNCIL OF ECONOMIC ADVISERS ISSUE BRIEF  
JANUARY 2017

---

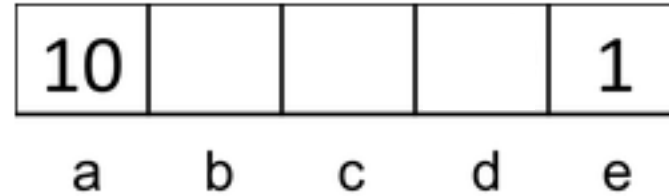
## DISCOUNTING FOR PUBLIC POLICY: THEORY AND RECENT EVIDENCE ON THE MERITS OF UPDATING THE DISCOUNT RATE

*Weighing benefits and costs that take place over time requires discounting those amounts to present value equivalents. This necessitates selecting a discount rate which can adjust for the fact that resources are more valuable today than in the future if consumers prefer to consume today rather than wait, or if firms could be earning a positive return on invested resources. Current guidance from the office of management and budget requires using both a 7 percent and 3 percent real discount rate in regulatory benefit-cost analyses. This issue brief reassesses the current choice of discount rates and methodologies for selecting the 3 percent and 7 percent rates. Empirical evidence suggests that real interest rates around the world have come down since the last evaluation of the rates, and new theoretical advances considering future uncertainty likely suggest lower long term rates, as well. In general the evidence supports lowering these discount rates, with a plausible best guess based on the available information being that the lower discount rate should be at most 2 percent while the upper discount rate should also likely be reduced.*

$\gamma =$   
0.93  
or  
0.97

# Quiz: Discounting

- Given:

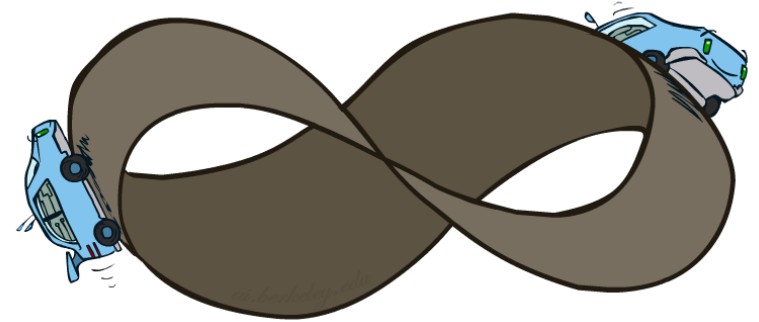


- Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic
- 
- Quiz 1: For  $\gamma = 1$ , what is the optimal policy?
  - Quiz 2: For  $\gamma = 0.1$ , what is the optimal policy?



# 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 ( $\pi$  depends on time left)

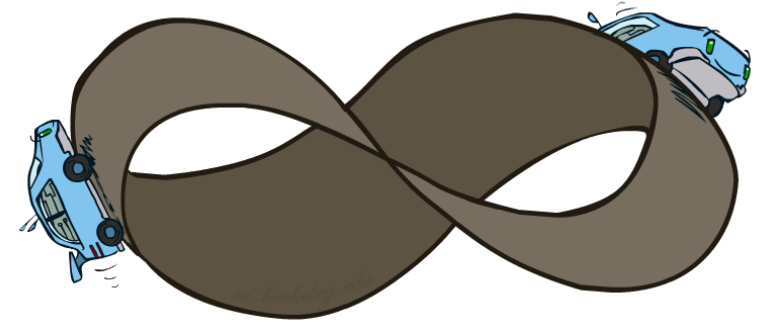


# Infinite Utilities?!

- 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 ( $\pi$  depends on time left)



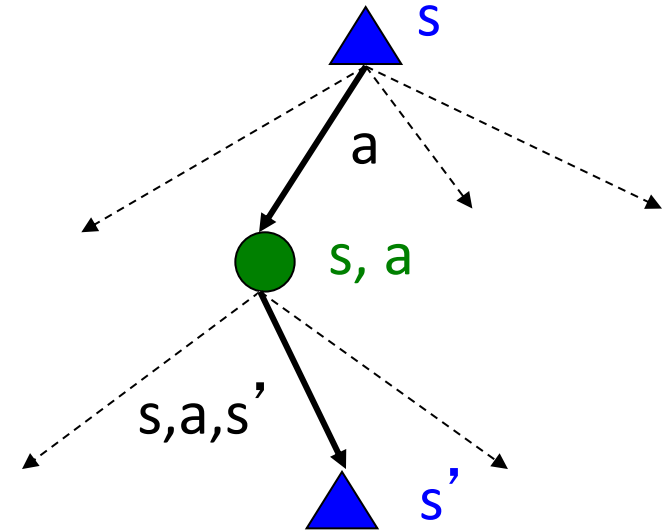
- Discounting: use  $0 < \gamma < 1$

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

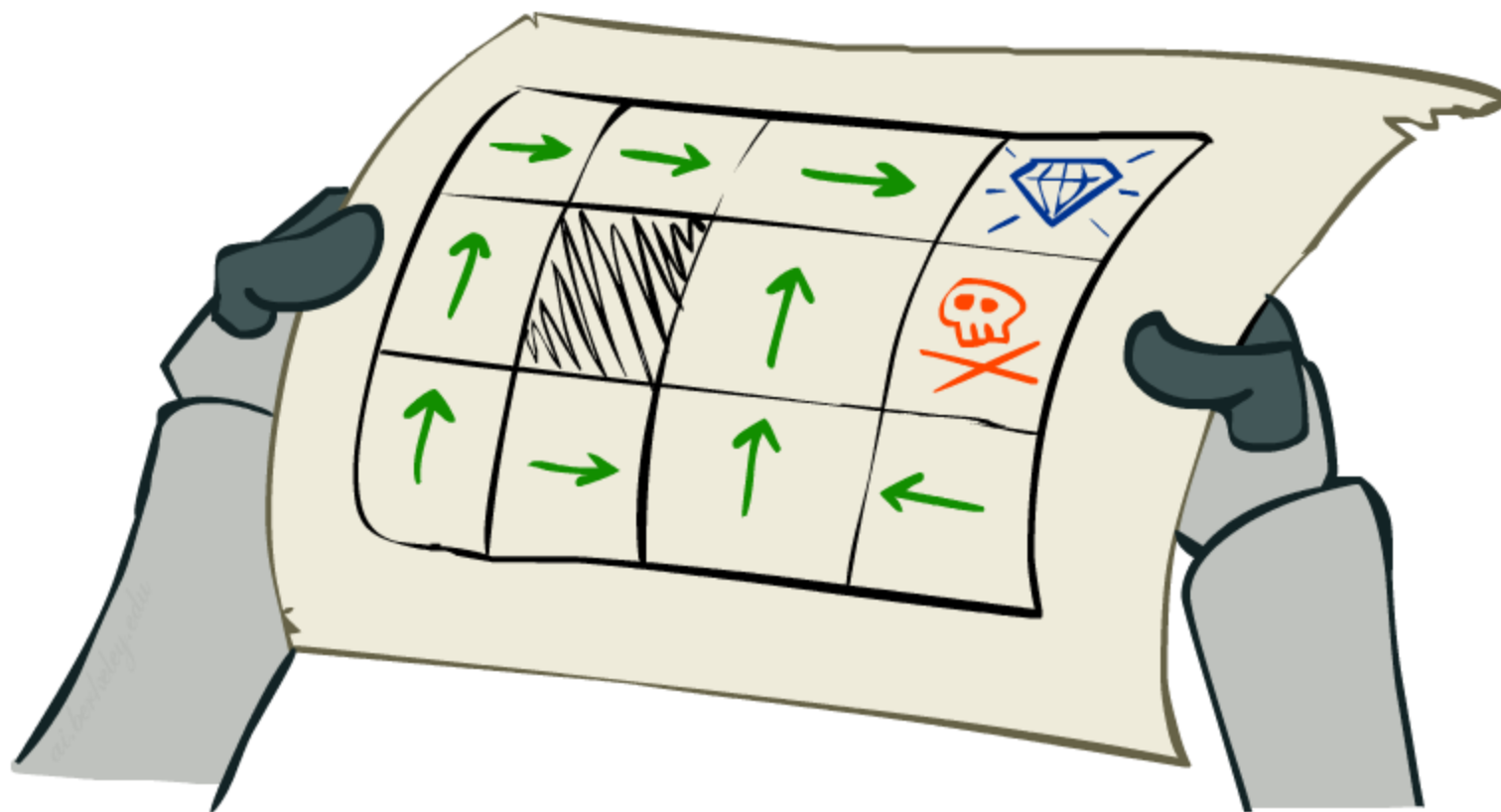
- Smaller  $\gamma$  means smaller “horizon” – shorter term focus
- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

# 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  $\gamma$ )
- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

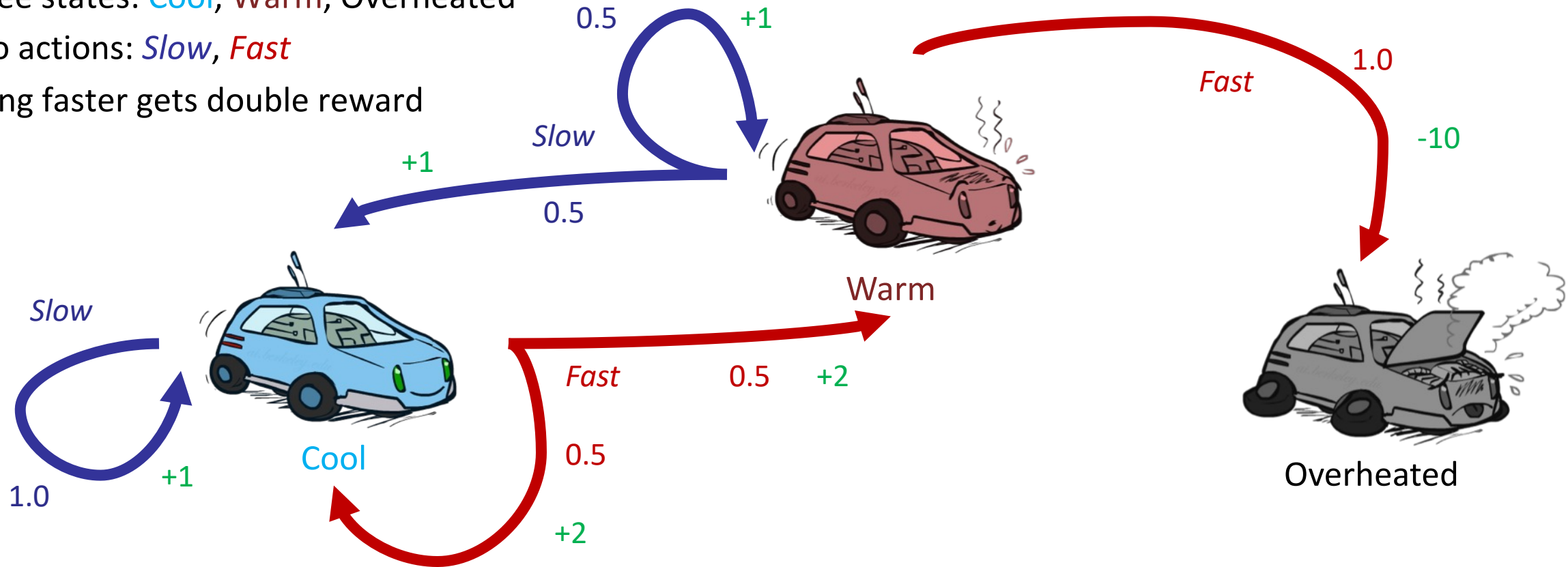


# Solving MDPs

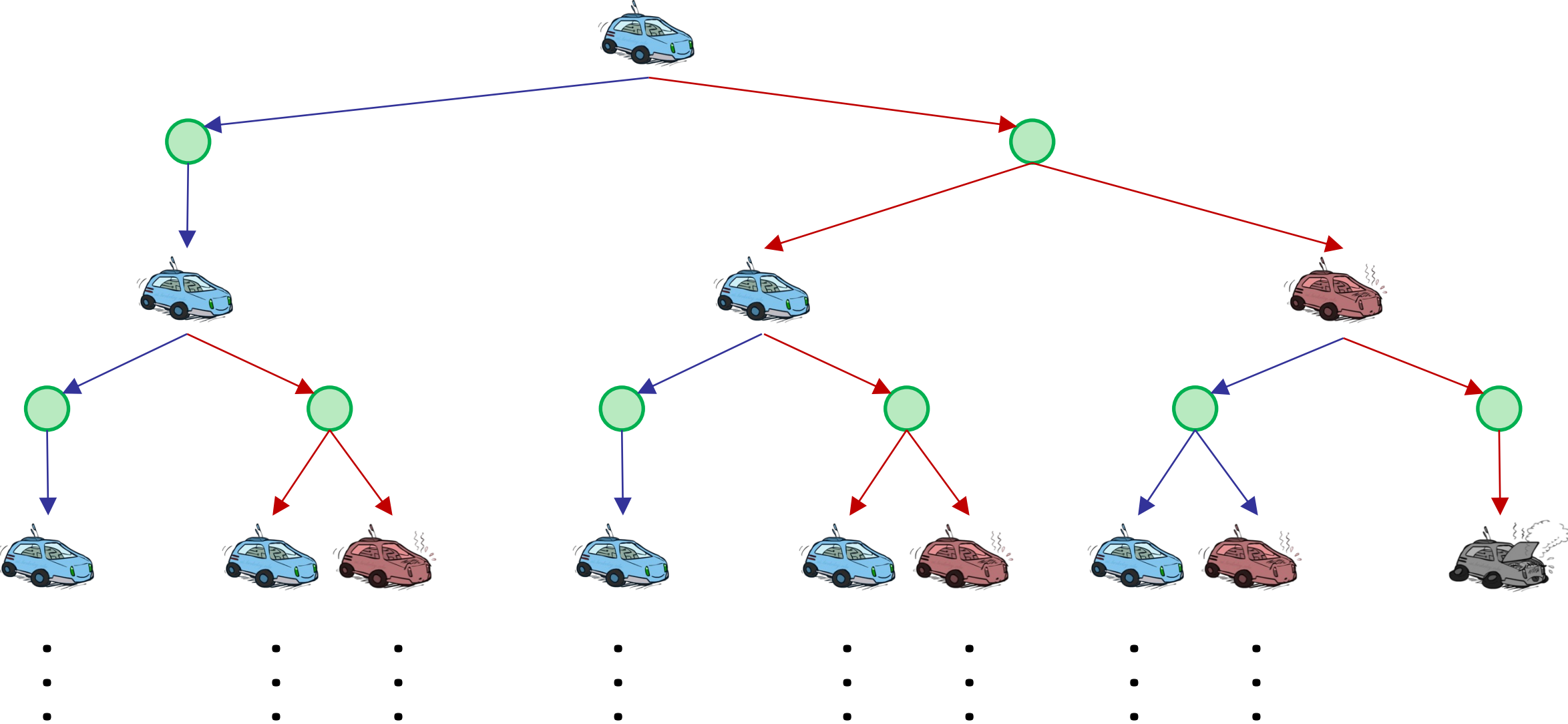


# Recall: Racing

- A robot car wants to travel far, quickly
- Three states: **Cool**, **Warm**, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward



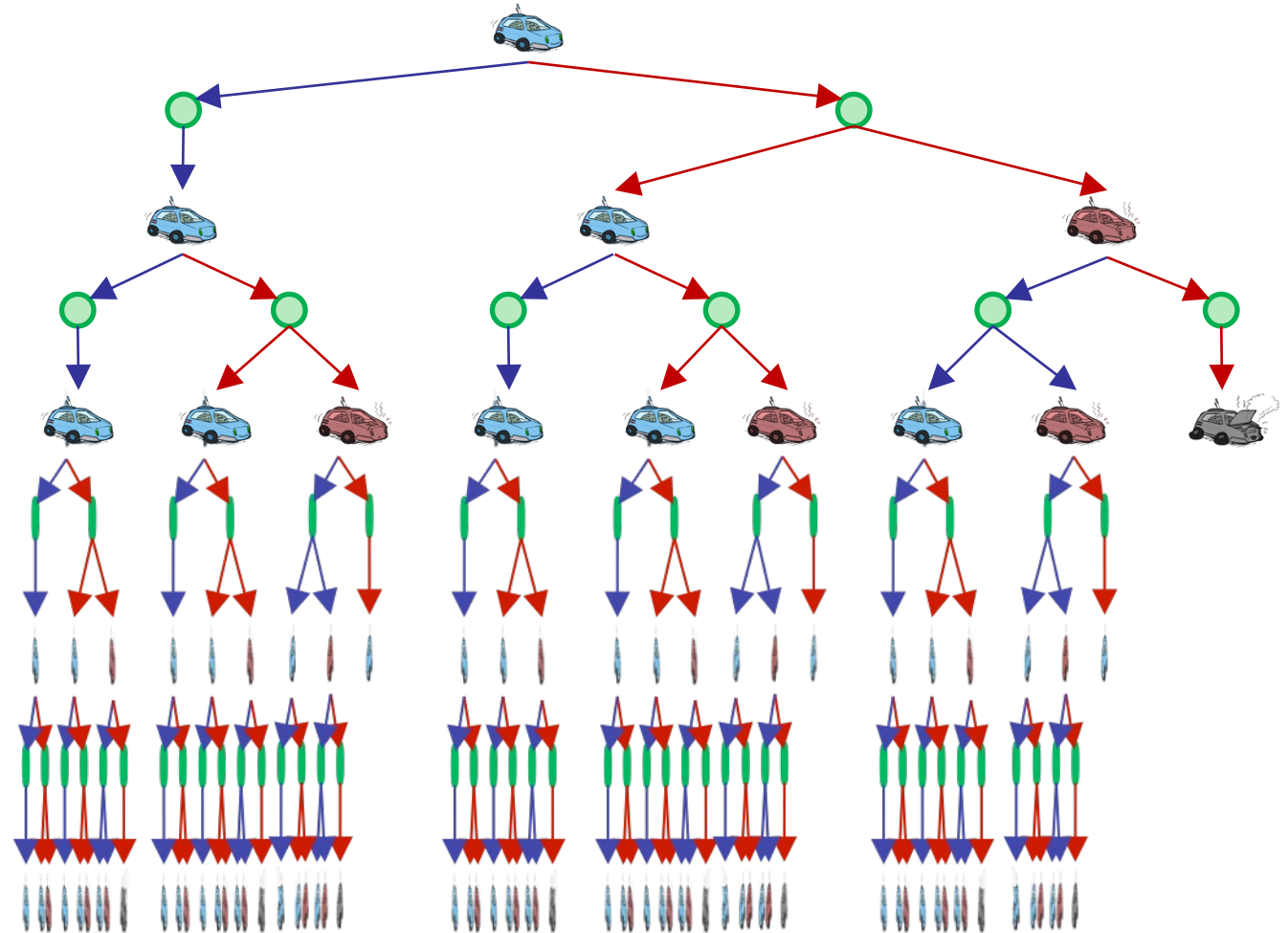
# Racing Search Tree





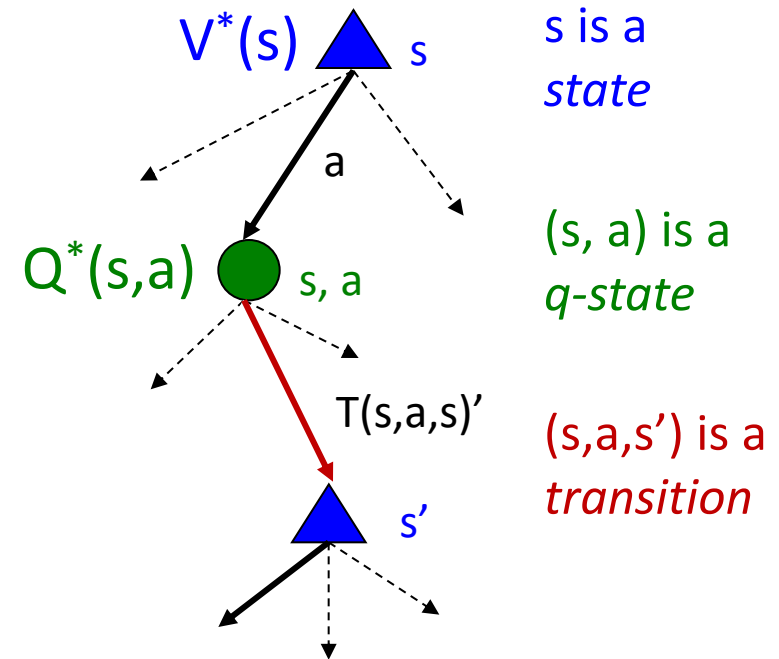
# Racing Search Tree

- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once, cache the rest in a lookup table
- 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  $\gamma < 1$



# Optimal Quantities

- The value (utility) of a state  $s$ :  
 $V^*(s)$  = expected utility starting in  $s$  and acting optimally
- The value (utility) of a q-state  $(s,a)$ :  
 $Q^*(s,a)$  = expected utility starting out having taken action  $a$  from state  $s$  and (thereafter) acting optimally
- The optimal policy:  
 $\pi^*(s)$  = optimal action from state  $s$



# Optimal Quantities in Gridworld

- The value (utility) of a state  $s$ :

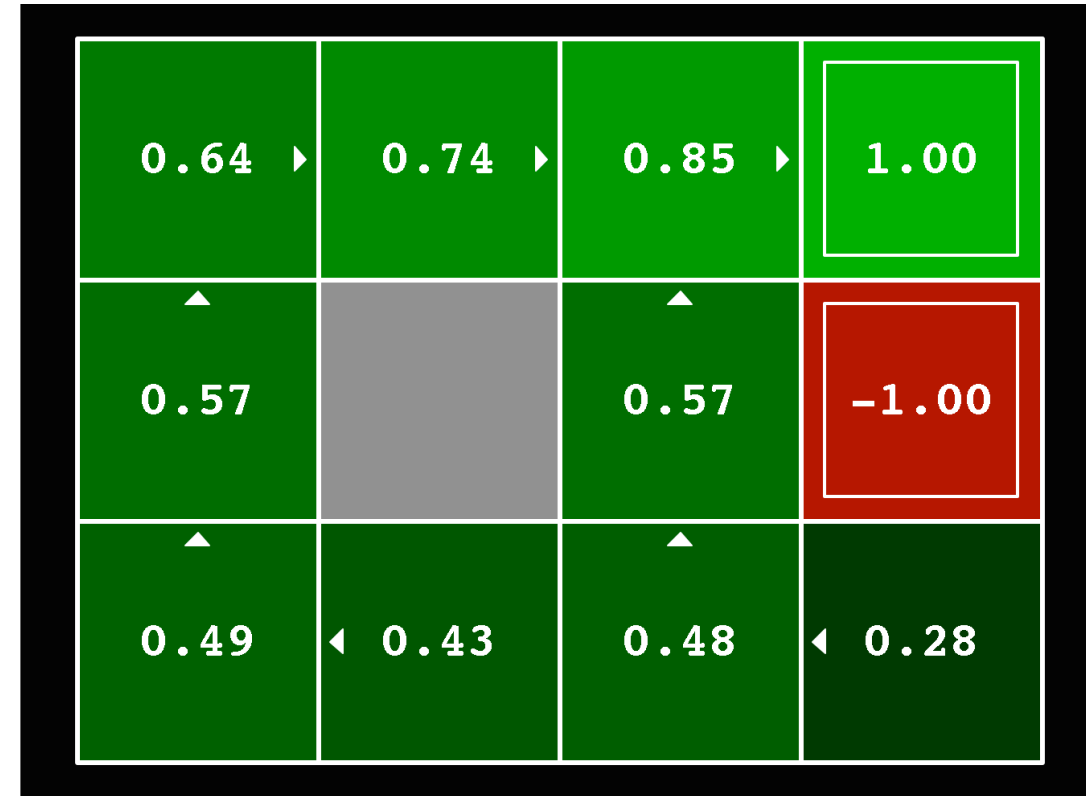
$V^*(s)$  = expected utility starting in  $s$  and acting optimally

- The value (utility) of a  $q$ -state  $(s,a)$ :

$Q^*(s,a)$  = expected utility starting out having taken action  $a$  from state  $s$  and (thereafter) acting optimally

- The optimal policy:

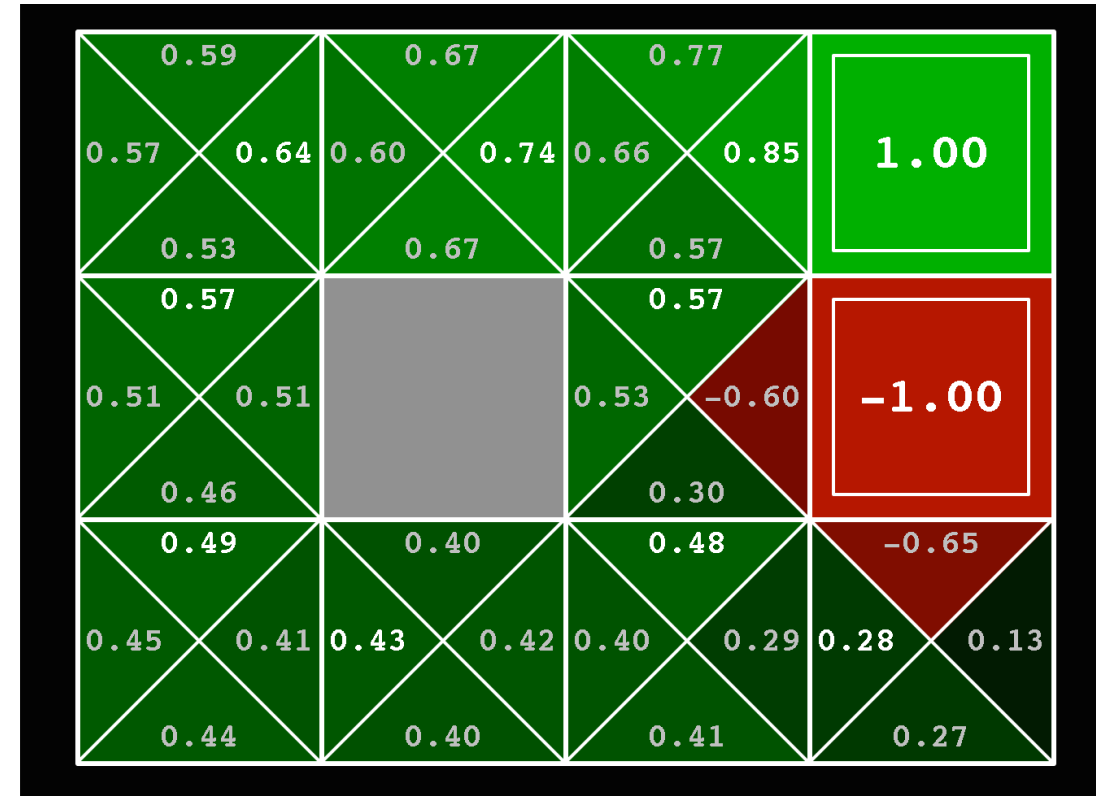
$\pi^*(s)$  = optimal action from state  $s$



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Optimal Quantities in Gridworld

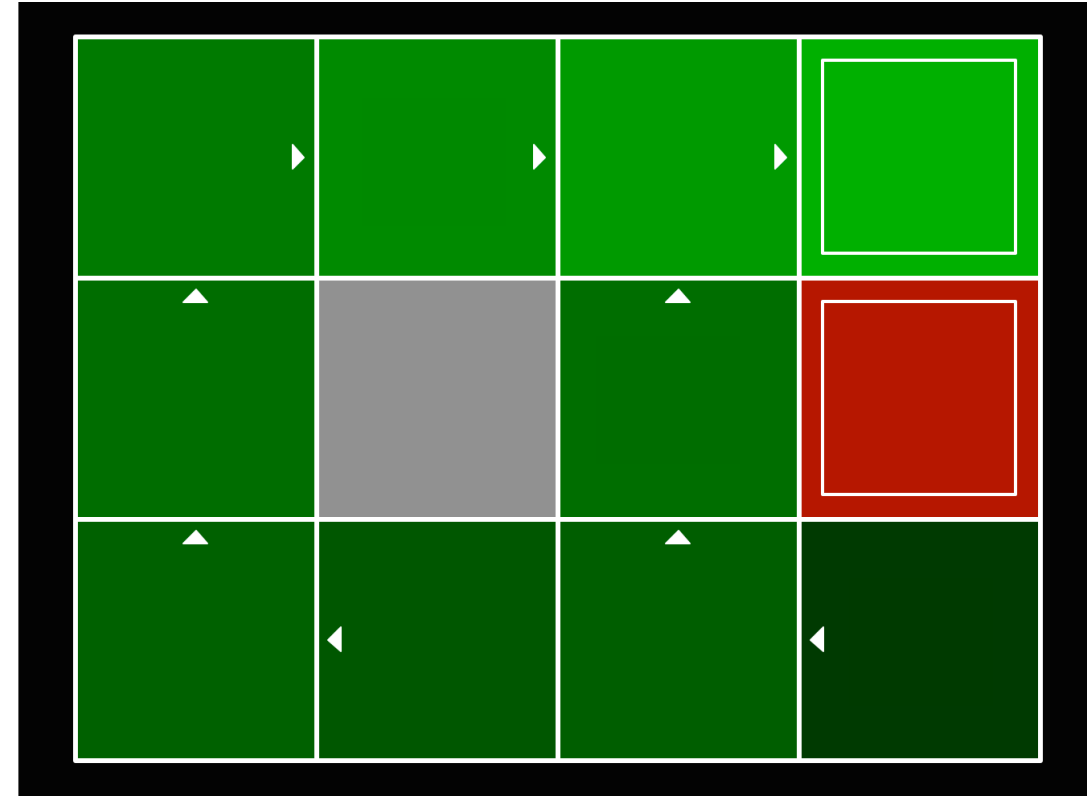
- The value (utility) of a state  $s$ :  
 $V^*(s)$  = expected utility starting in  $s$  and acting optimally
- The value (utility) of a q-state  $(s,a)$ :  
 $Q^*(s,a)$  = expected utility starting out having taken action  $a$  from state  $s$  and (thereafter) acting optimally
- The optimal policy:  
 $\pi^*(s)$  = optimal action from state  $s$



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Optimal Quantities in Gridworld

- The value (utility) of a state  $s$ :  
 $V^*(s)$  = expected utility starting in  $s$  and acting optimally
- The value (utility) of a q-state  $(s,a)$ :  
 $Q^*(s,a)$  = expected utility starting out having taken action  $a$  from state  $s$  and (thereafter) acting optimally
- The optimal policy:  
 $\pi^*(s)$  = optimal action from state  $s$



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Values of States

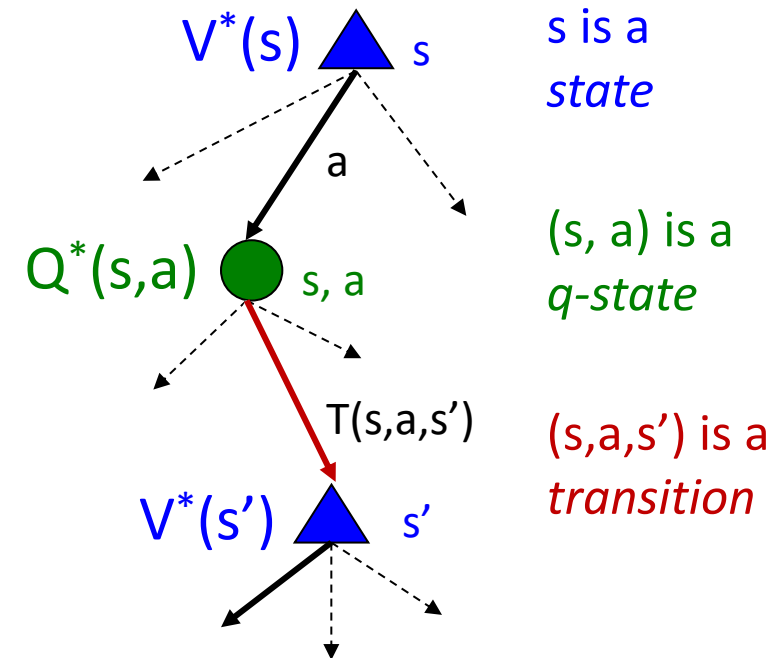
- Recursive definition of value (similar to expectimax):

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

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

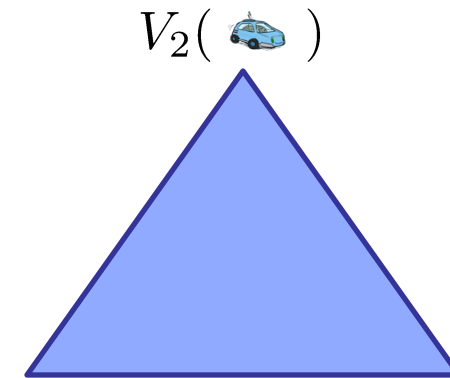
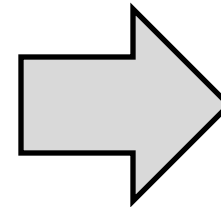
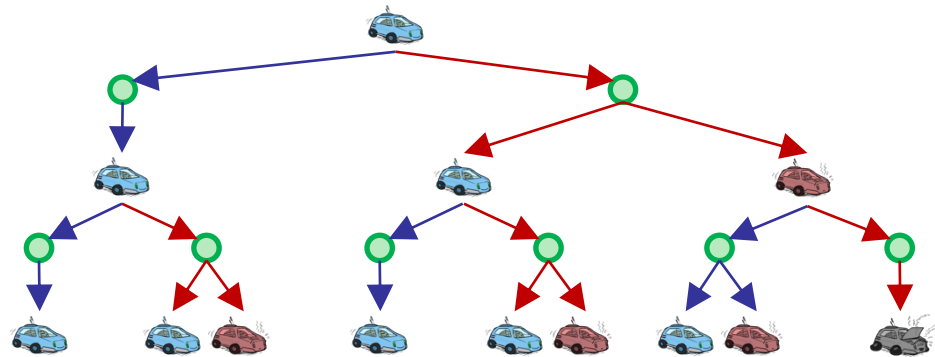
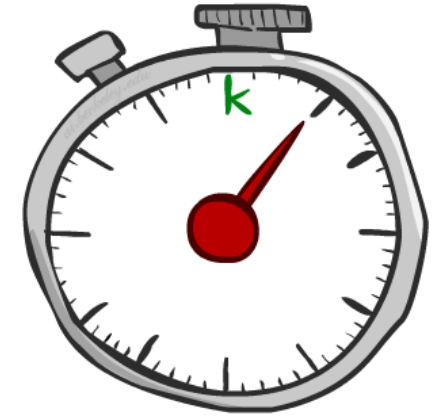
$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

- But how do we solve these equations?

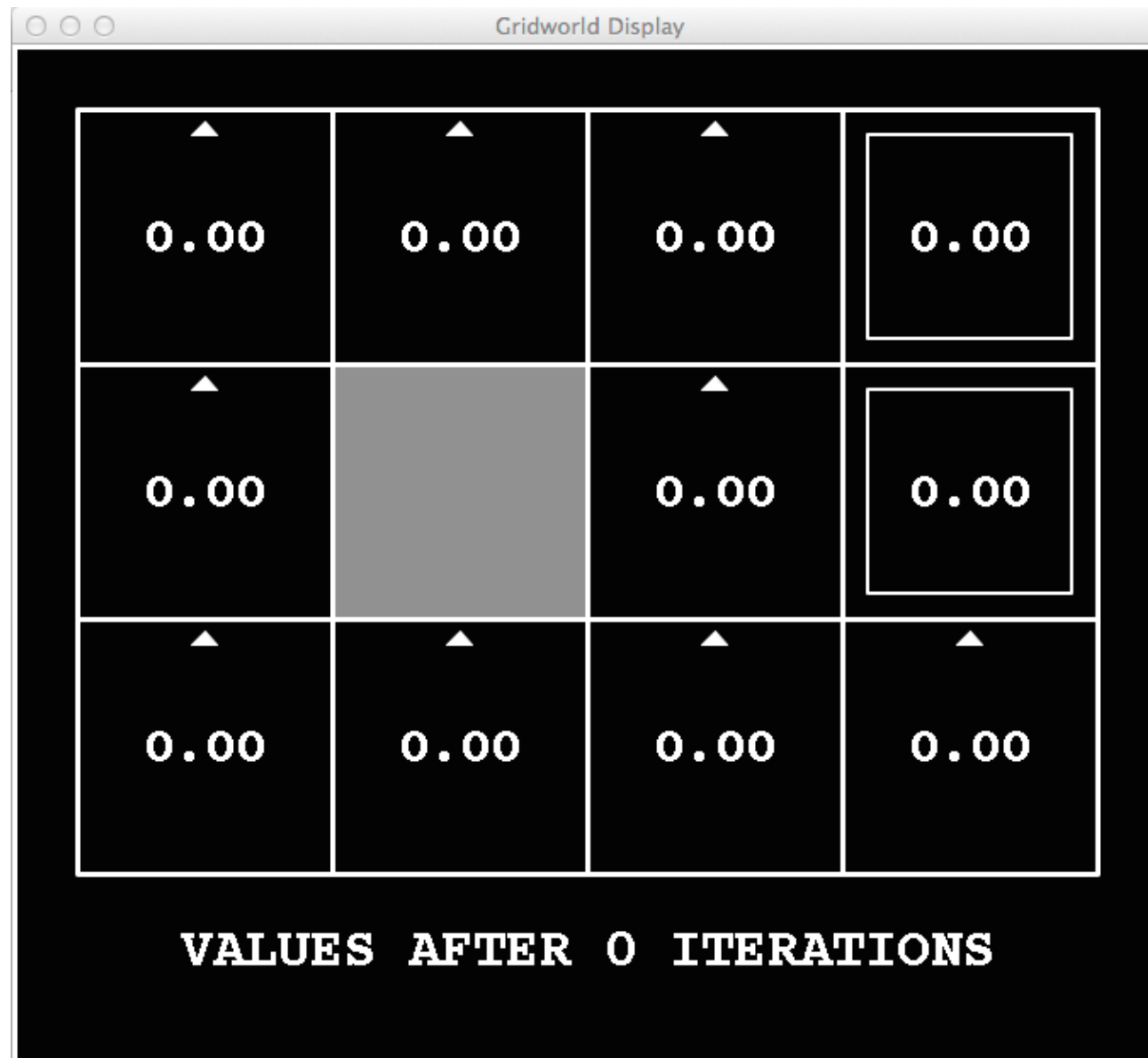


# Time-Limited Values

- 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  $s$



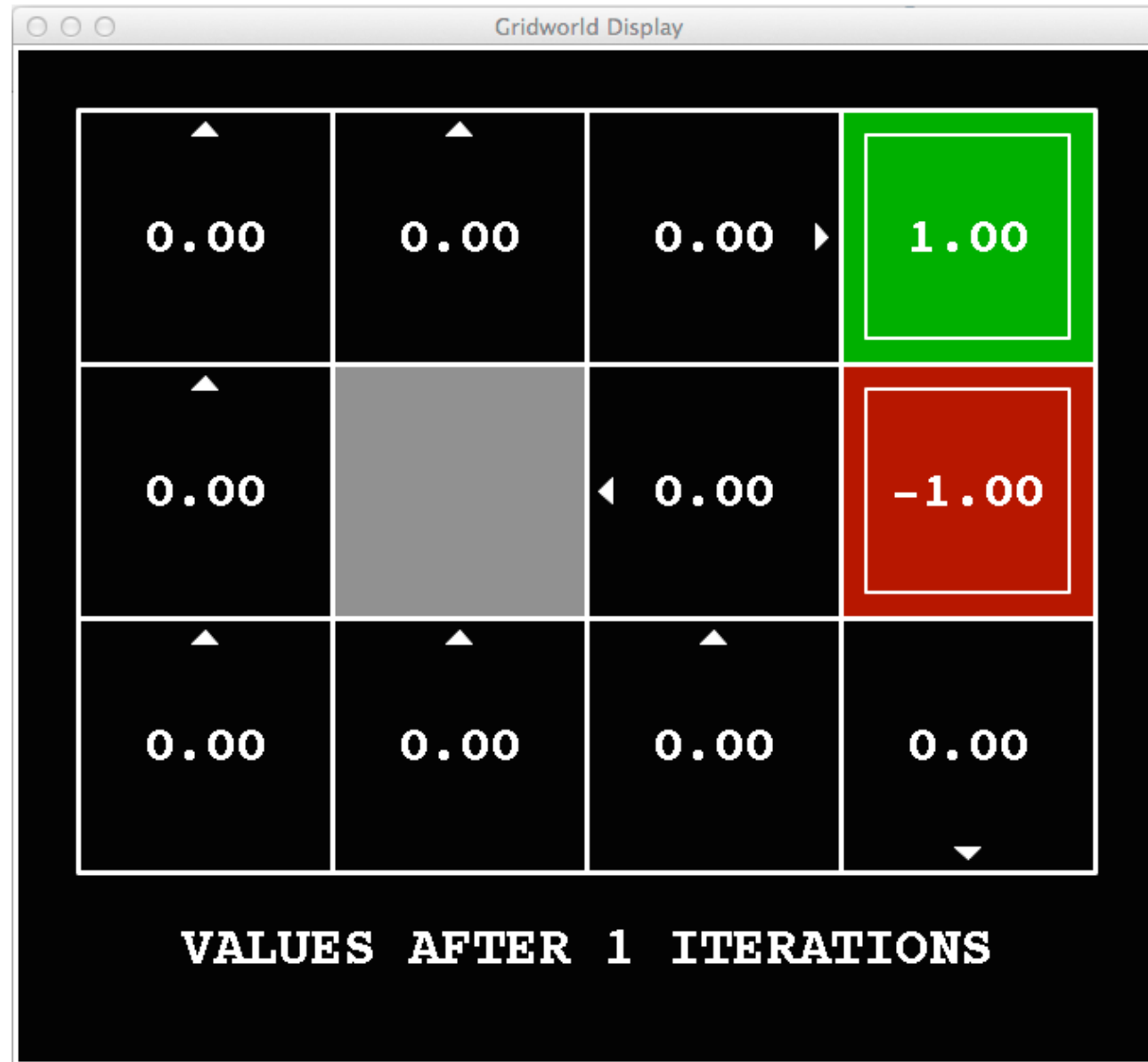
# k=0



Noise = 0.2  
Discount = 0.9  
Living reward = 0



# k=1



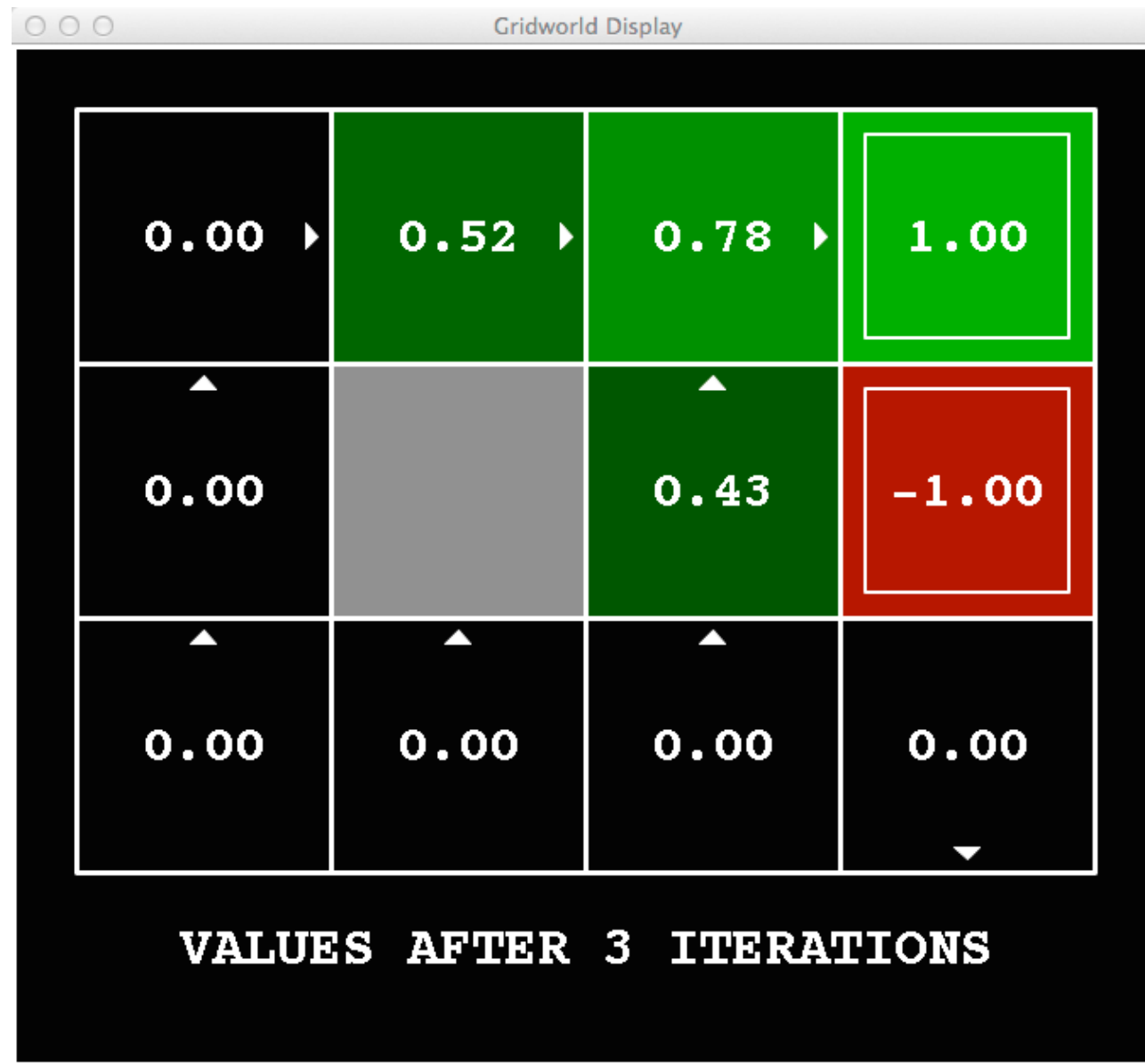
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=2



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=3



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=4



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=5



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=6



Noise = 0.2  
Discount = 0.9  
Living reward = 0

k=7



# k=8



Noise = 0.2  
Discount = 0.9  
Living reward = 0



# k=9



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=10



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=11



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=12



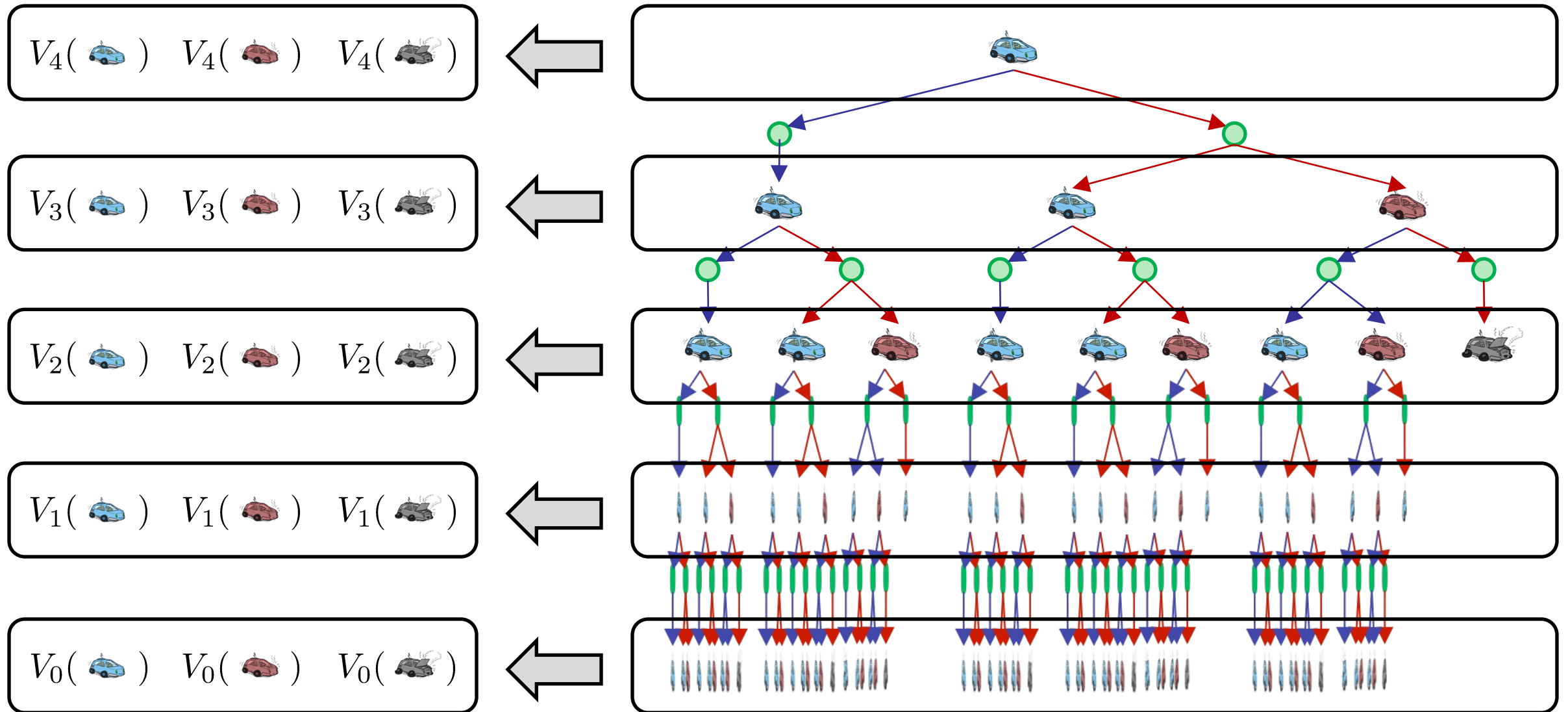
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=100



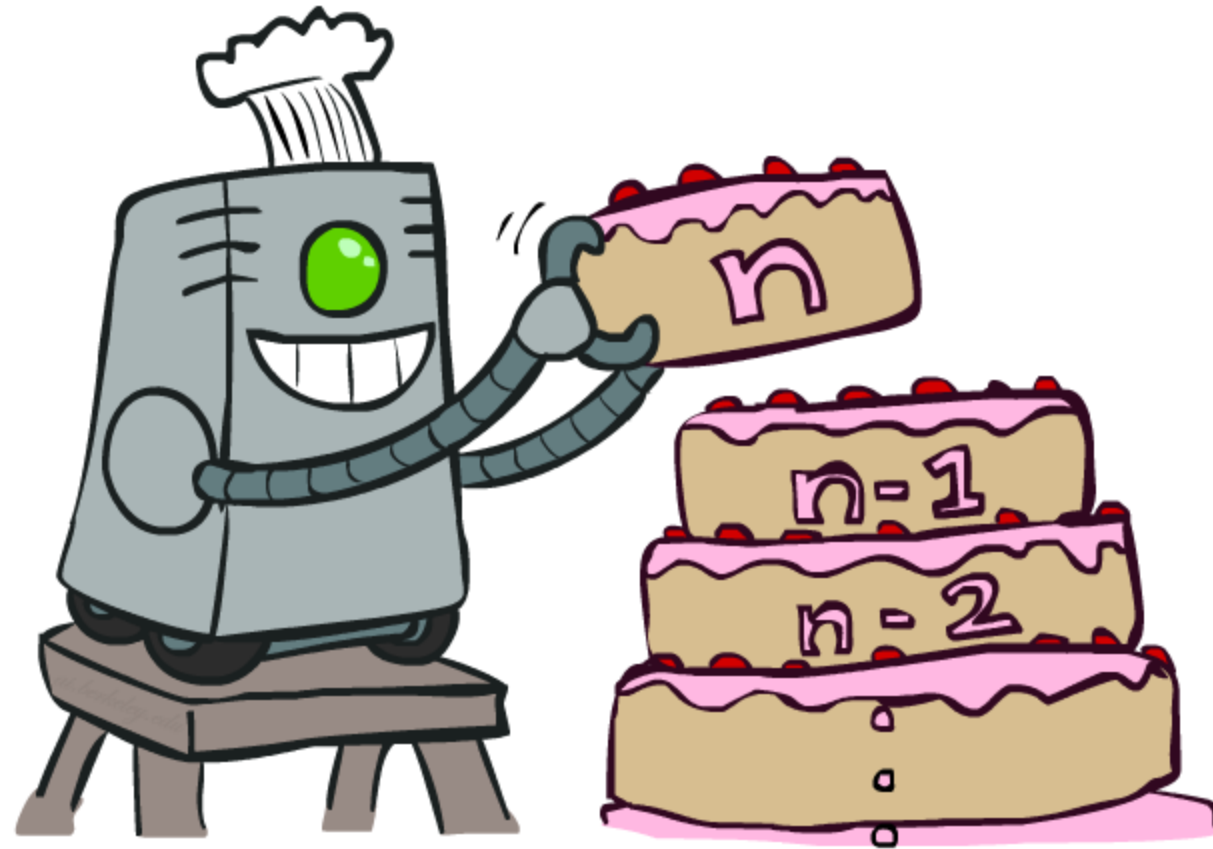
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Computing Time-Limited Values



# Value Iteration

---

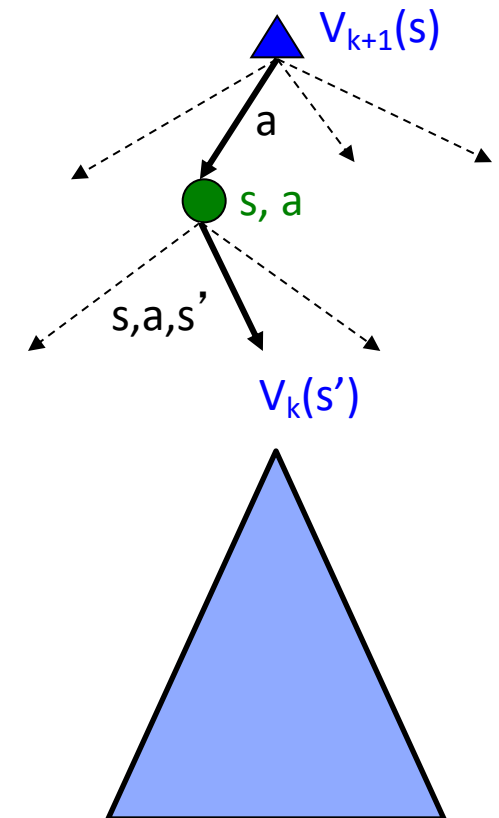


# Value Iteration

- 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 step of expectimax from each state:

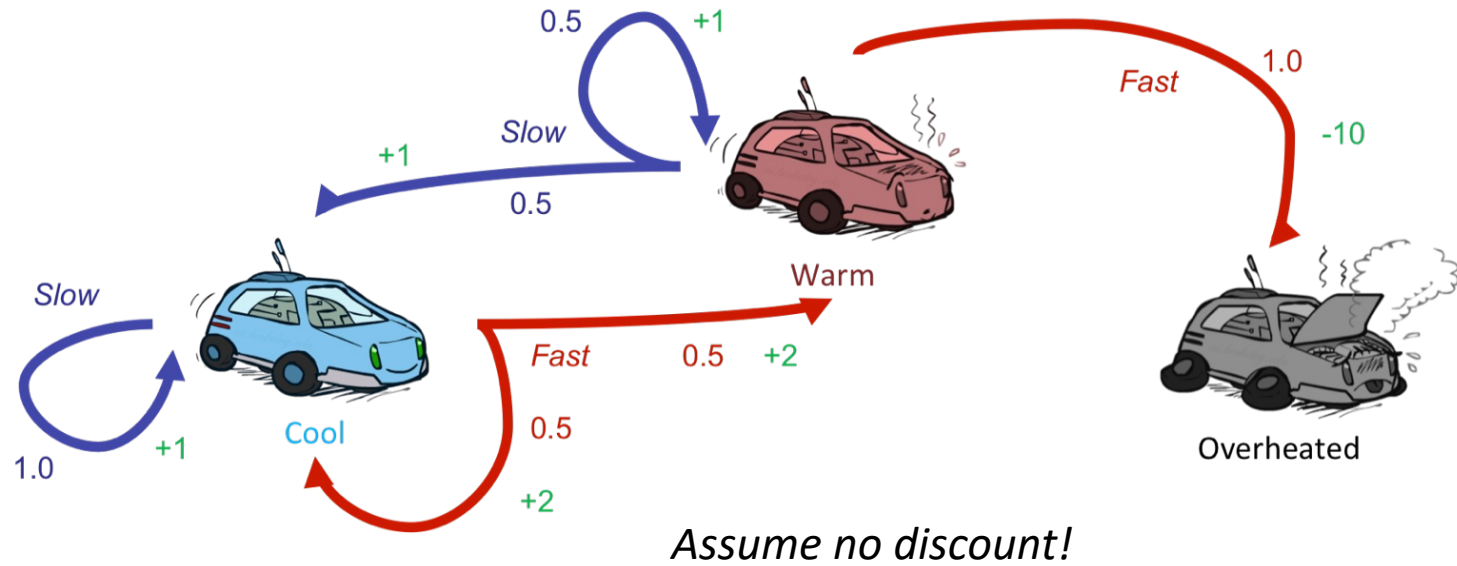
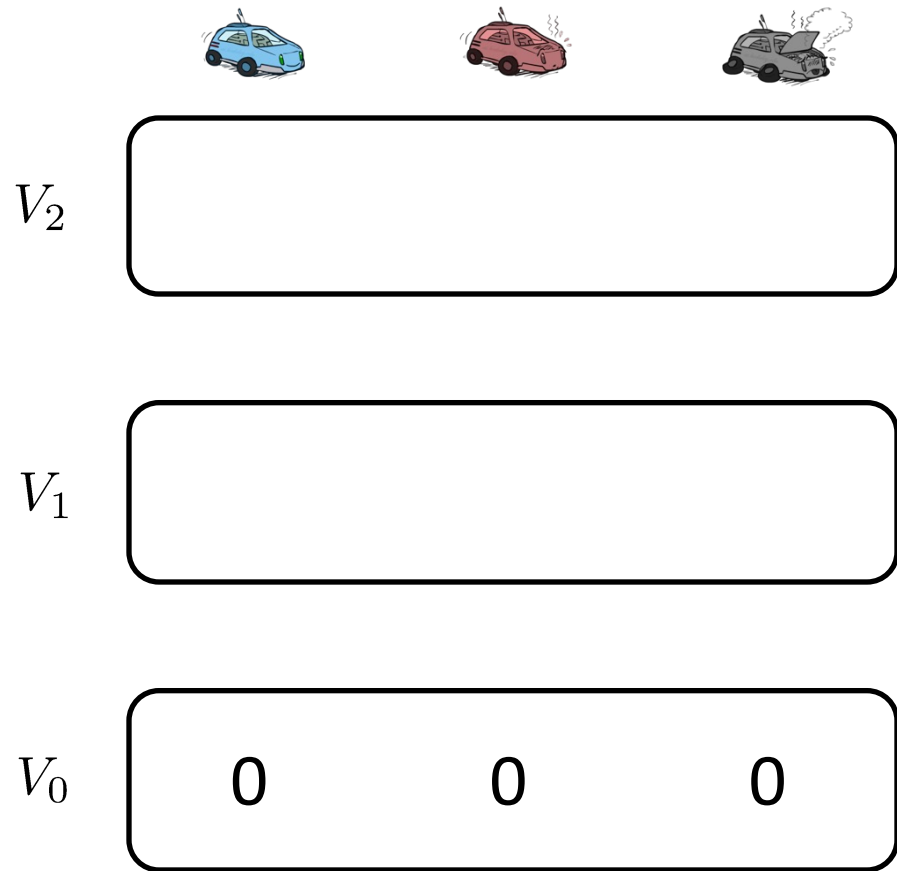
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

- Repeat until convergence, which yields  $V^*$
- 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



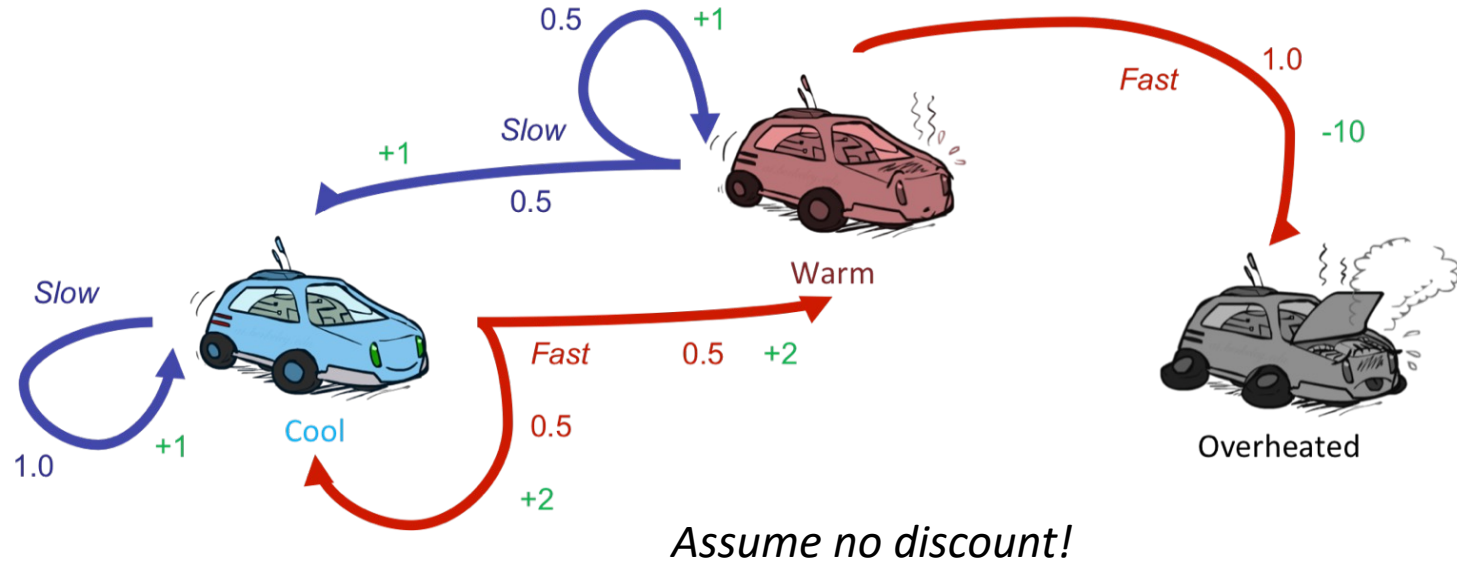
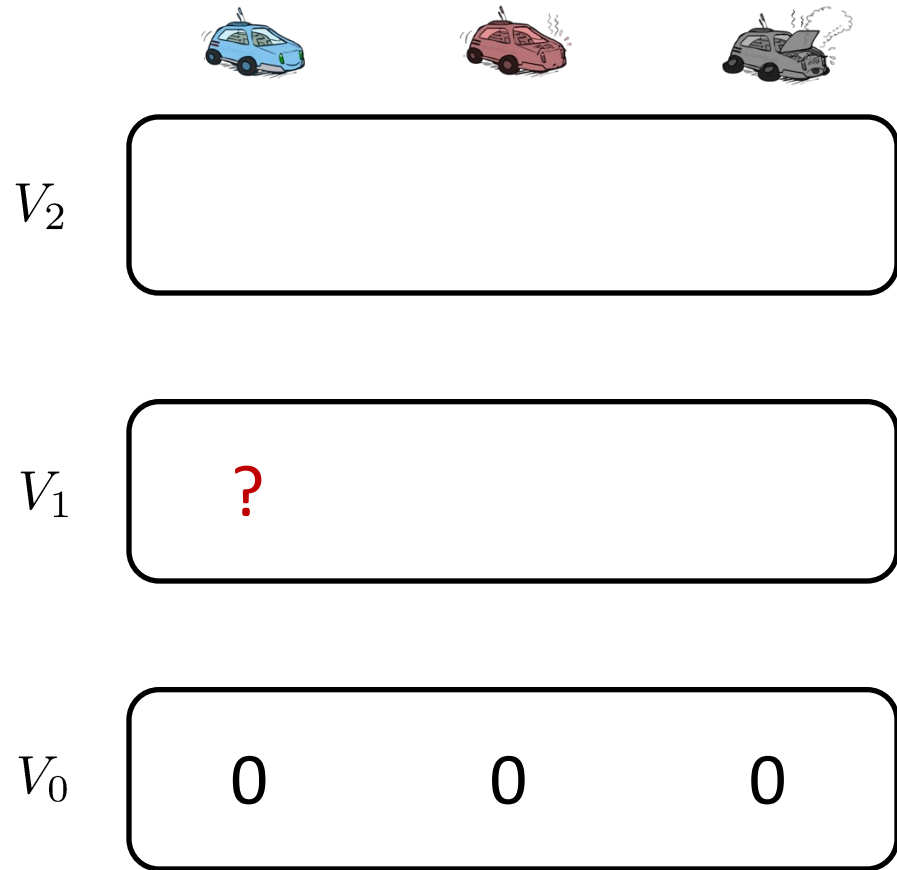


# Example: Value Iteration



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

# Example: Value Iteration






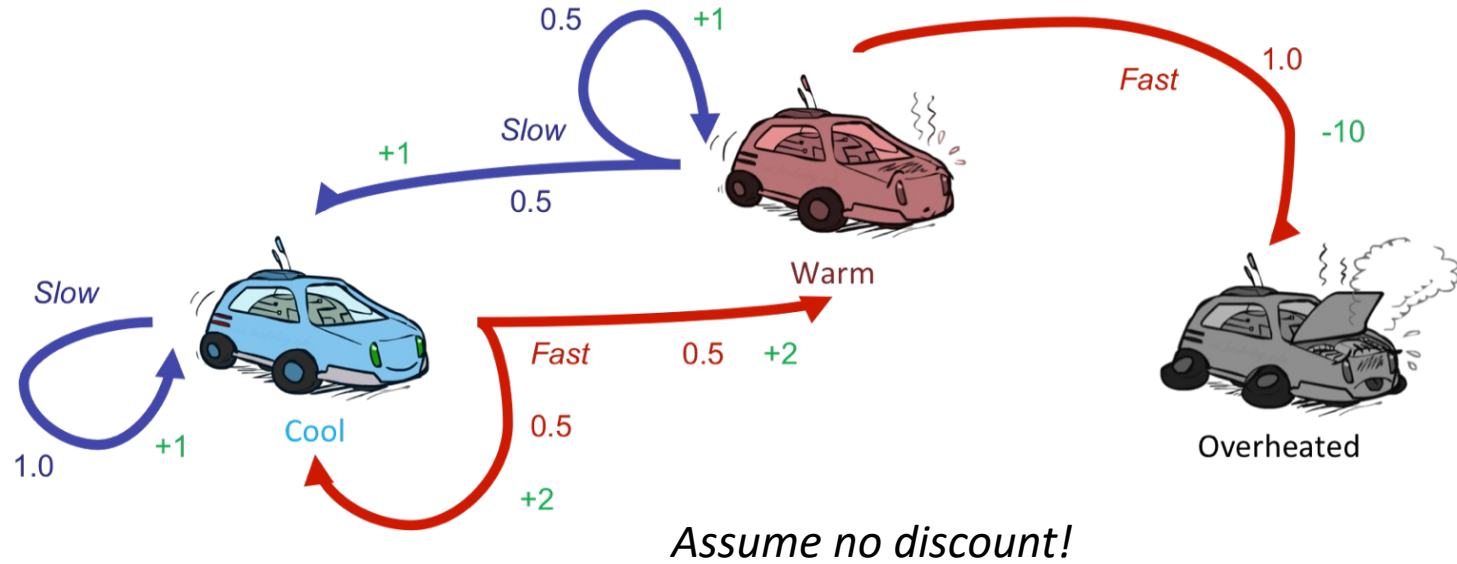
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

a=slow:  $1(1 + 0) = 1$

a=fast:  $0.5(2 + 0) + 0.5(2 + 0) = 2$

# Example: Value Iteration

			
$V_2$			
$V_1$	2	?	
$V_0$	0	0	0






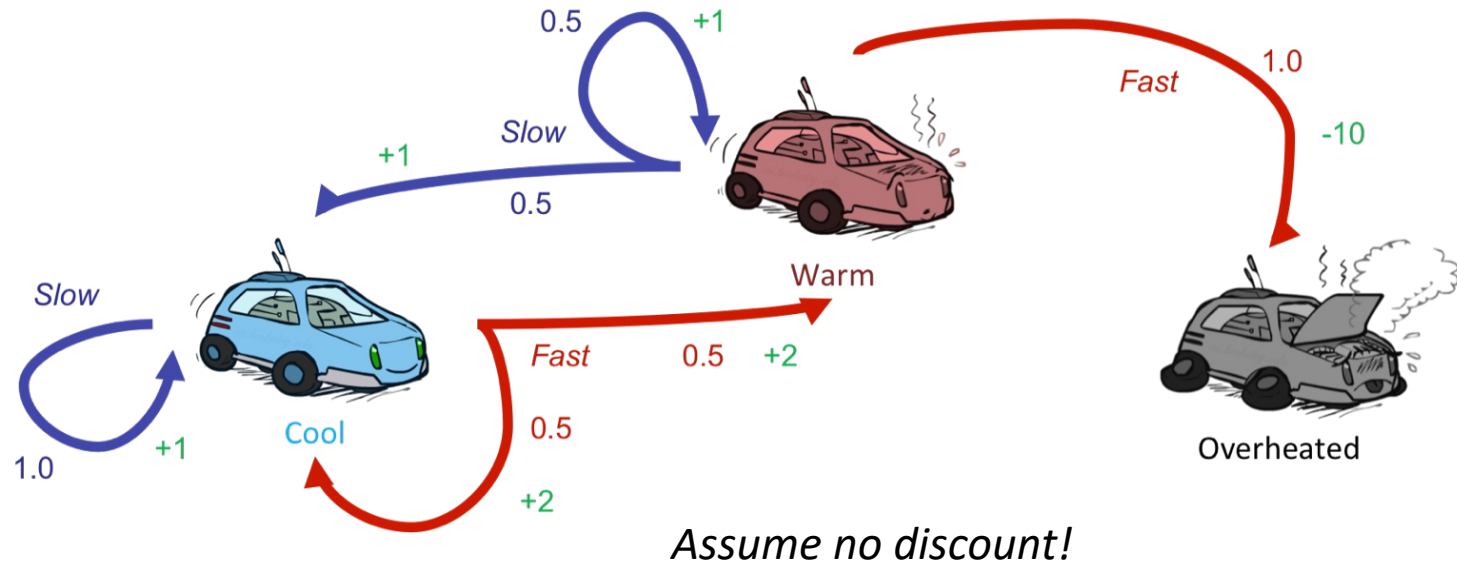
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

a=slow:  $0.5(1 + 0) + 0.5(1 + 0) = 1$

a=fast:  $1(-10 + 0) = -10$

# Example: Value Iteration

			
$V_2$	?		
$V_1$	2	1	0
$V_0$	0	0	0






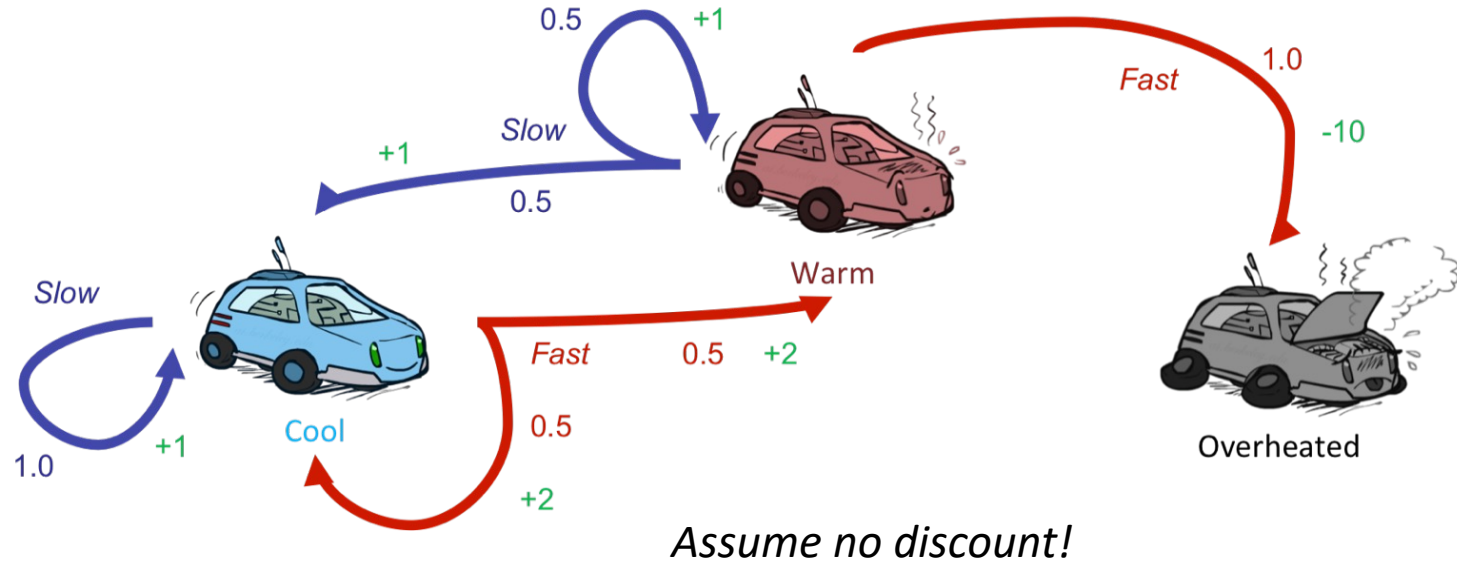
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

a=slow:  $1(1 + 2) = 3$

a=fast:  $0.5(2 + 2) + 0.5(2 + 1) = 3.5$

# Example: Value Iteration

			
$V_2$	3.5	?	0
$V_1$	2	1	0
$V_0$	0	0	0






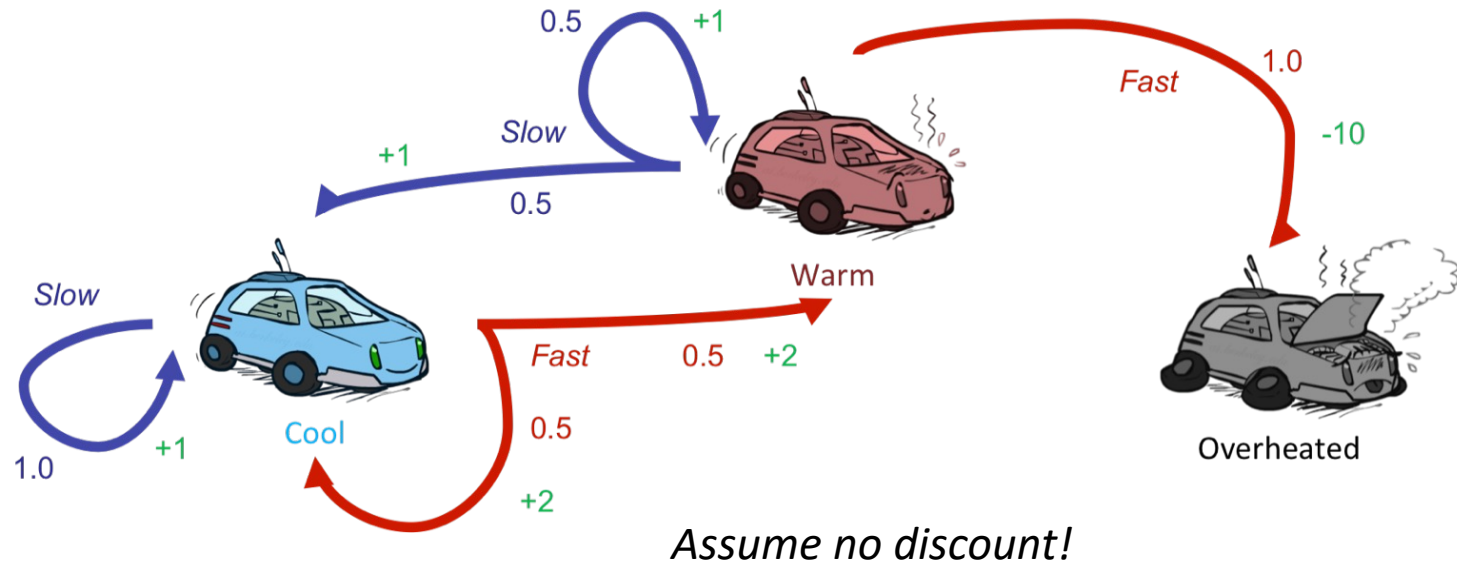
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

a=slow:  $0.5(1 + 2) + 0.5(1 + 1) = 2.5$

a=fast:  $1(-10 + 0) = -10$

# Example: Value Iteration

			
$V_2$	3.5	2.5	0
$V_1$	2	1	0
$V_0$	0	0	0



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

# What we did today

- Working with stochastic environments (but world model known)
- Introduced *MDPs* (describe problem) and *policies* (solution)
  - MDPs look similar to expectimax search trees
- Discussed how to solve MDPs
  - Optimal *state value*  $V^*(s)$ , and *q-state value*  $Q^*(s,a)$  are key quantities
  - *Bellman equation* characterizes the optimal value function:

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

- A key equation in RL and this class!
- *Value iteration* is an algorithm to solve the Bellman equation

# Next Time: Policy-Based Methods

---