

# Announcement

---

- Homework 4
  - Due: May 3, 11:59pm

# Markov Decision Processes

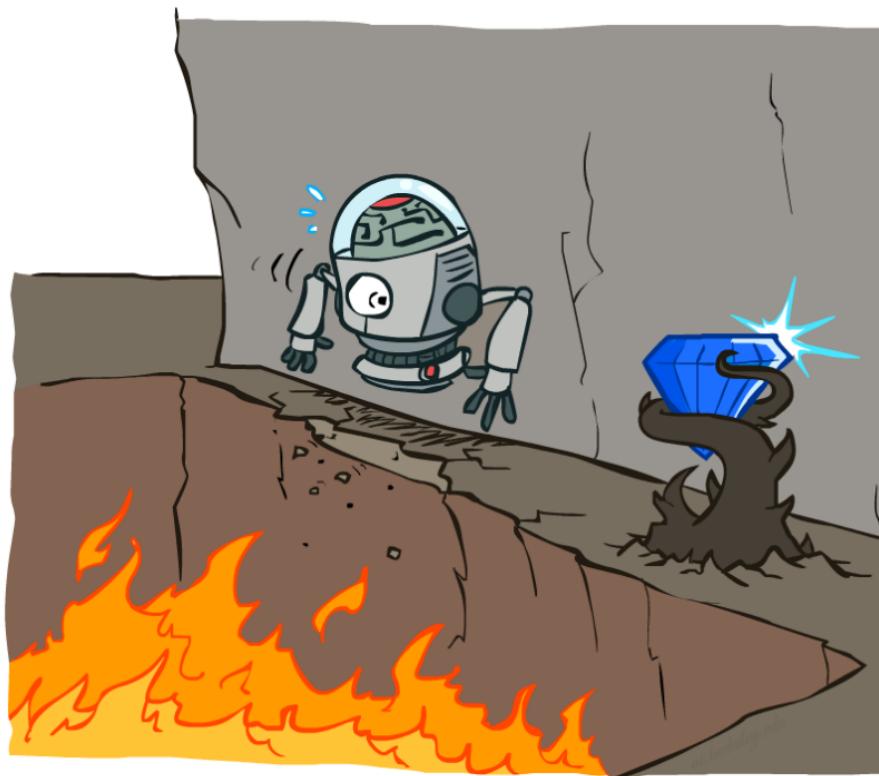


AIMA Chapter 17



[Adapted from slides by Dan Klein and Pieter Abbeel at UC Berkeley]

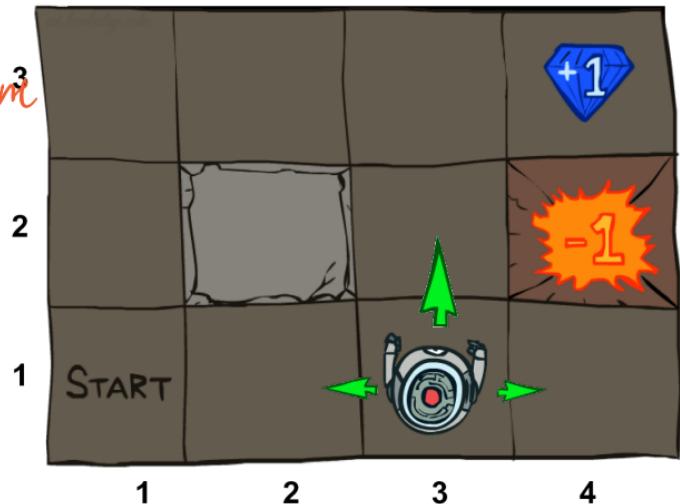
# Non-Deterministic Search



# Example: Grid World

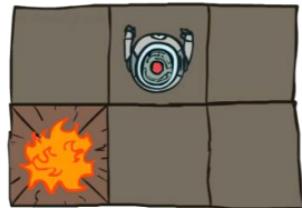
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is 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

1. Not as planned  
2. reward / punishment



# Grid World Actions

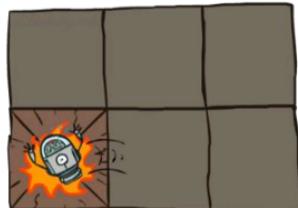
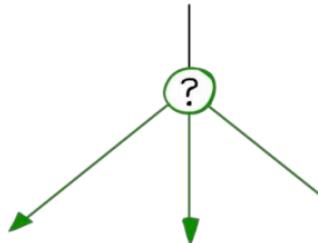
Deterministic Grid World



Stochastic Grid World

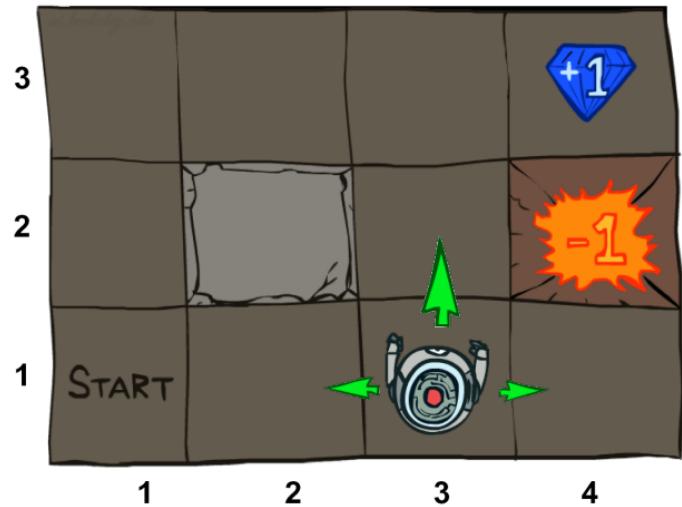


?



# 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  $T(s, a, s') = P(s' | s, a)$ 
    - 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  $R(s')$  *some time inde from actions.*
  - 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?

When present observed past and future  
Inde

- “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

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

Sign =

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

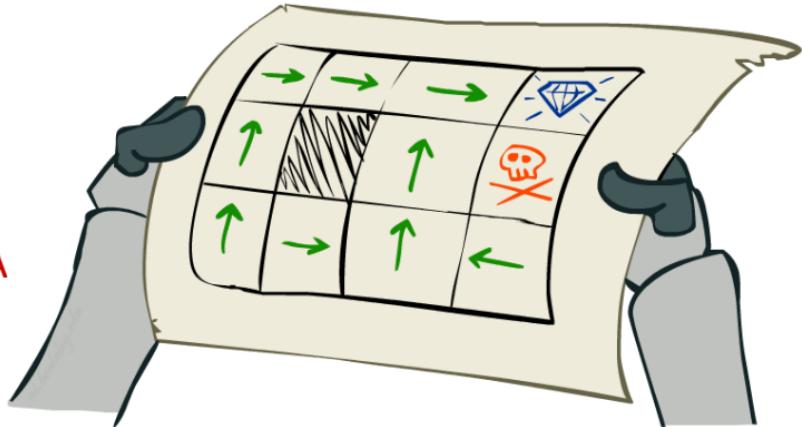


Andrey Markov  
(1856-1922)

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

# Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal  
"Rule"
- 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

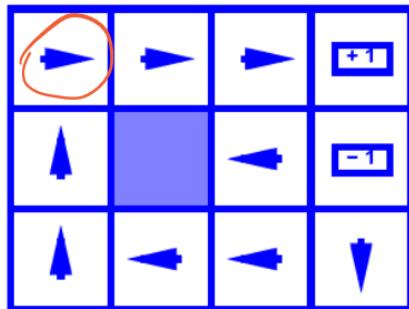


how to find  
best action.

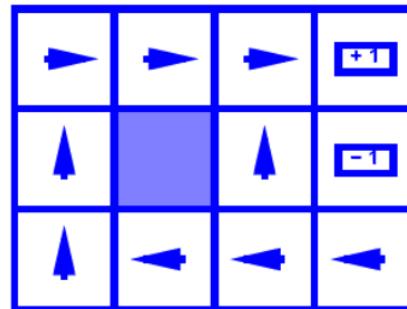
# Optimal Policies

*choice at start:*

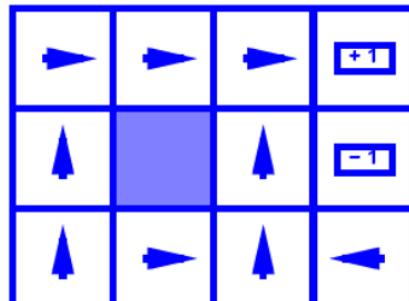
$R(s)$  = "living reward"



$$R(s) = -0.01$$

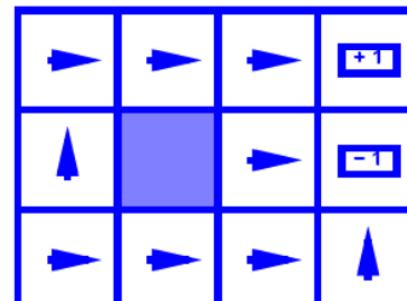


$$R(s) = -0.03$$



$$R(s) = -0.4$$

*find rewards*

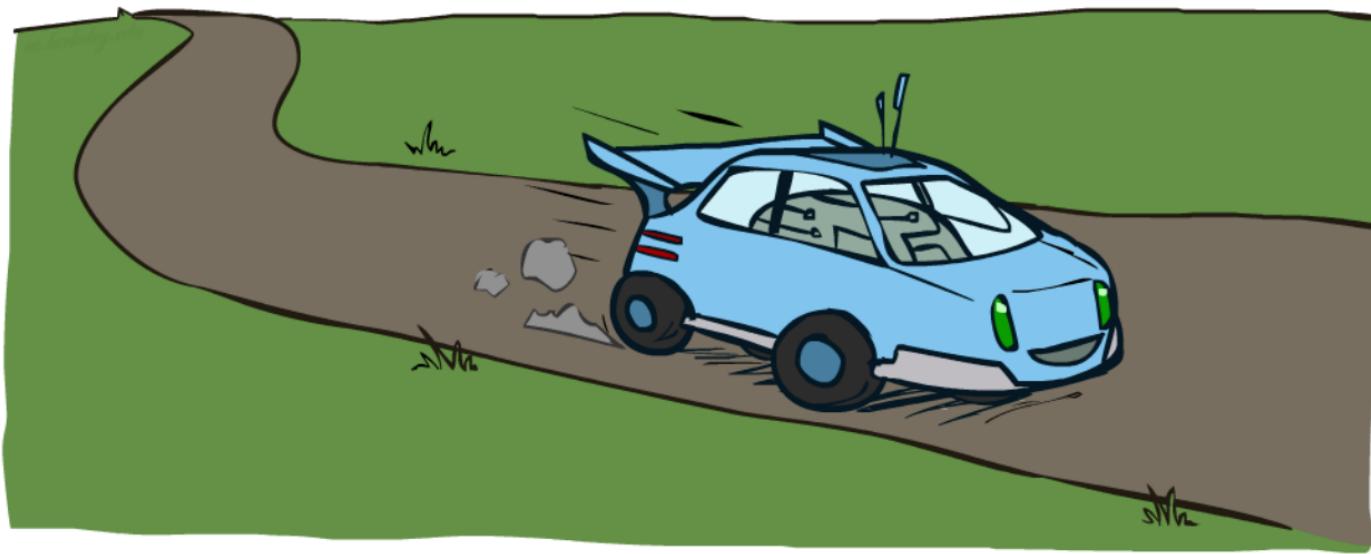


$$R(s) = -2.0$$

*spending first*

# Example: Racing

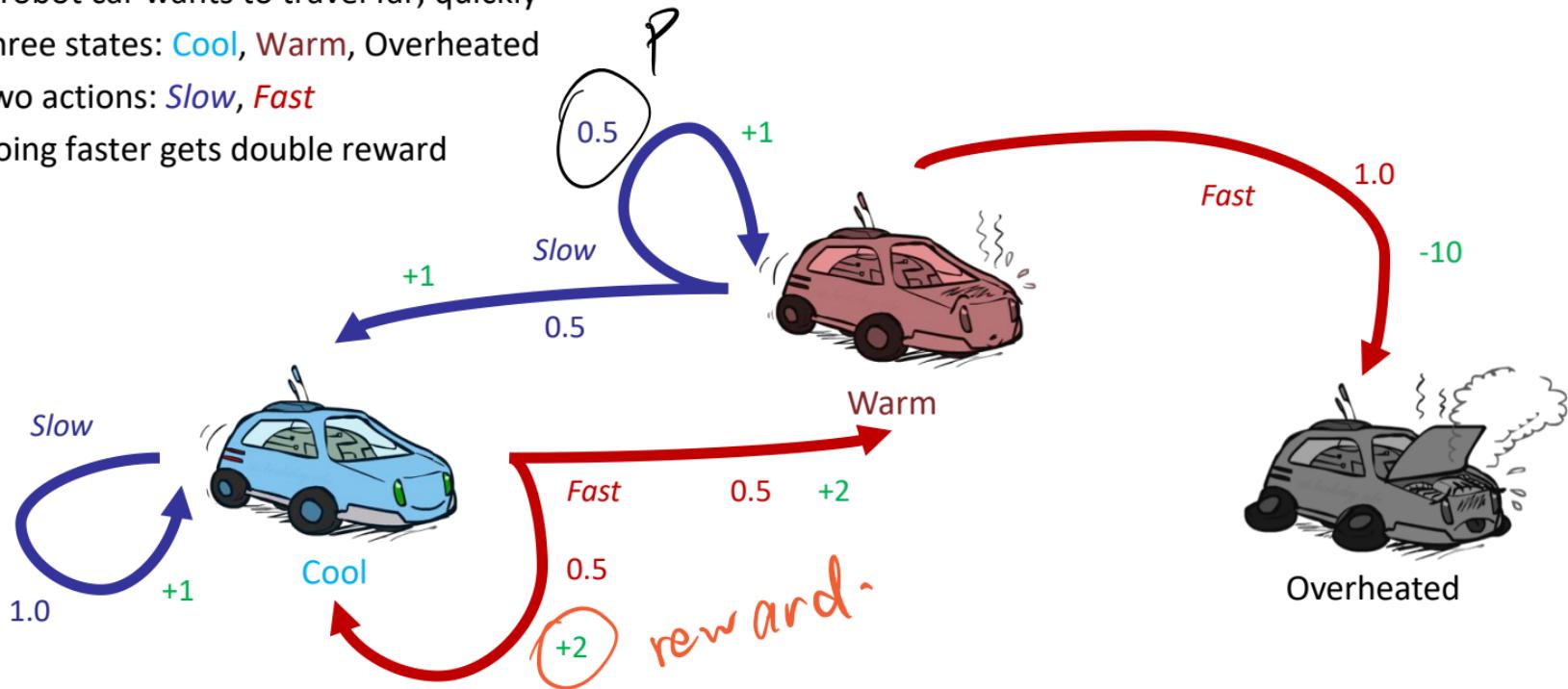
Goal  
|  
|  
Reward



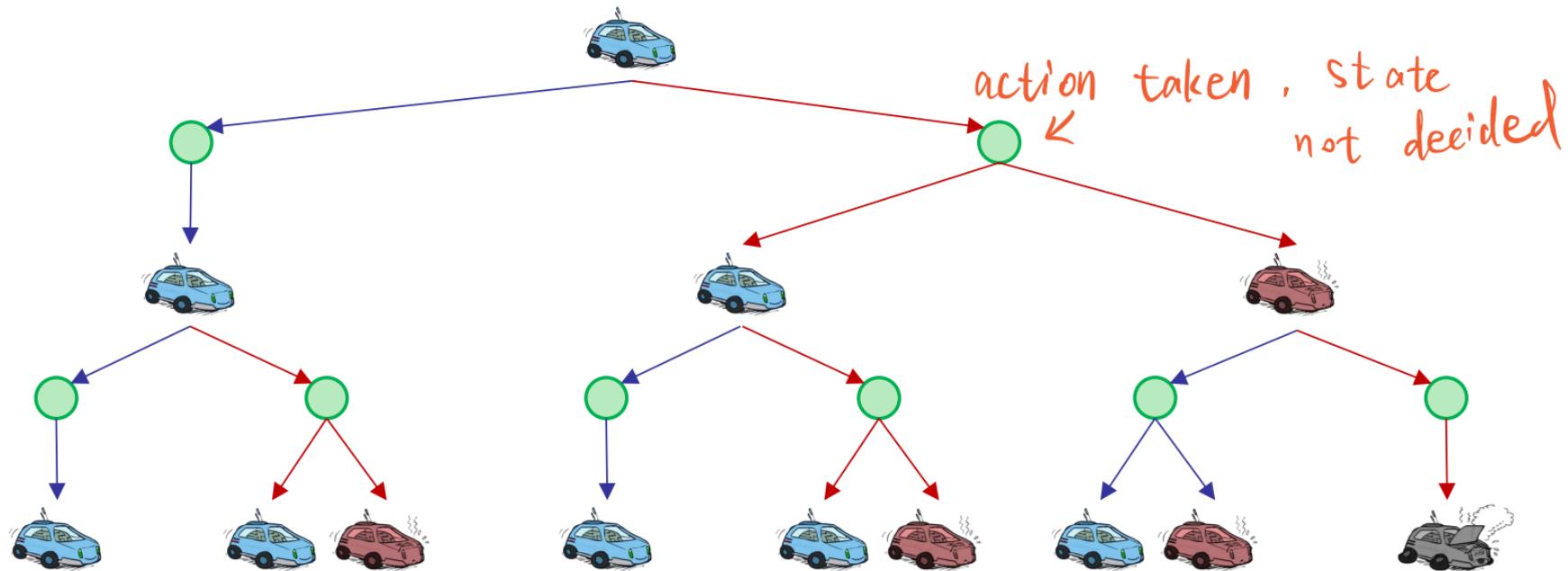
# FSM

## Example: Racing

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

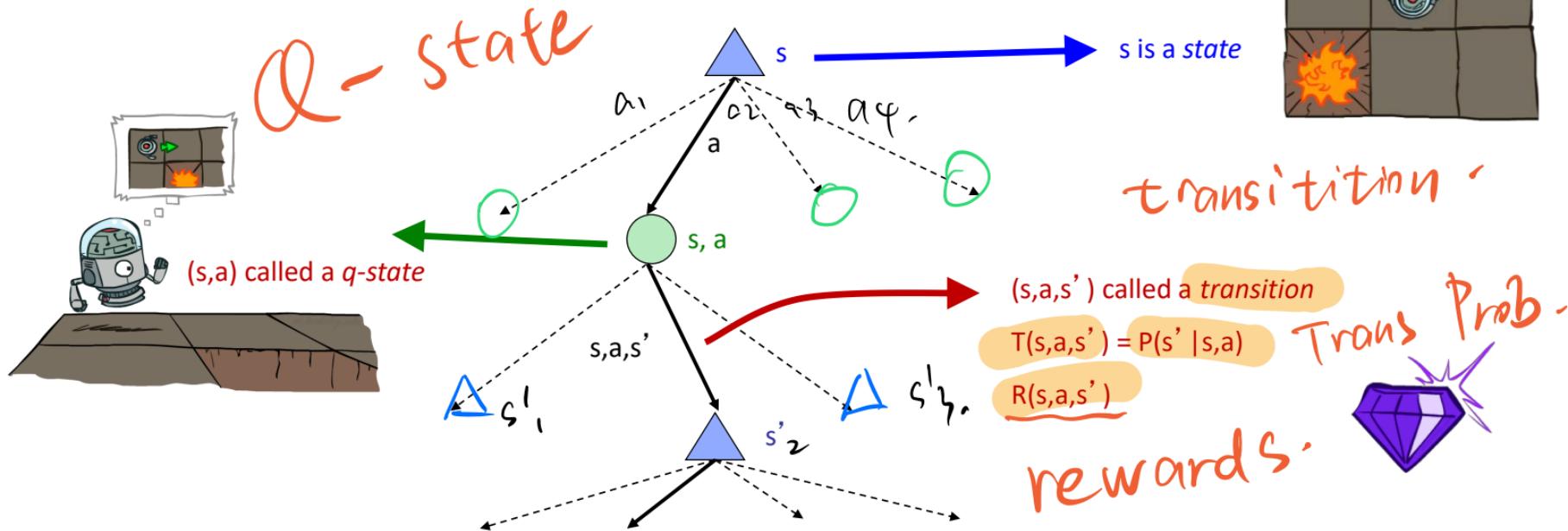


# Racing Search Tree

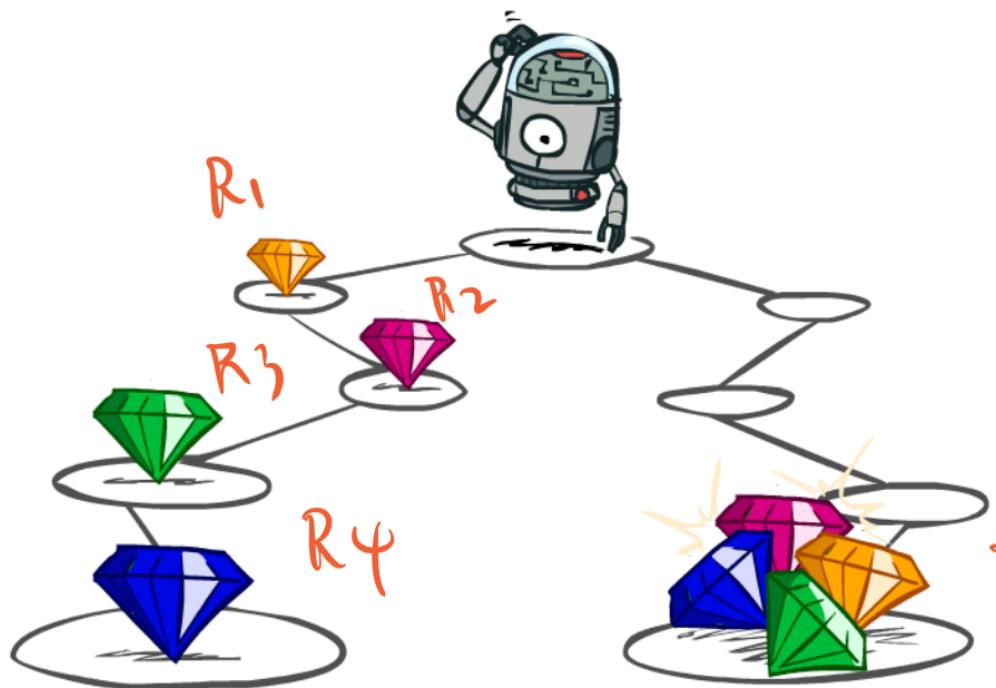


# MDP Search Trees

- Each MDP state projects an expectimax-like search tree



# Utilities of Sequences



Which better?

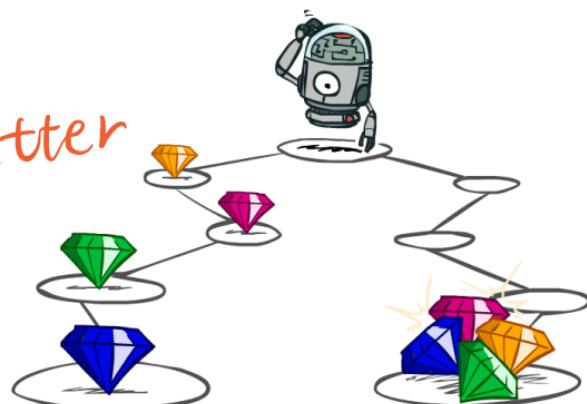
# Utilities of Sequences

- 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       $[1, 0, 0]$

maybe ... better

speed !

faster



# 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

time ↑ weight ↓

reward

# Discounting

- How to discount?

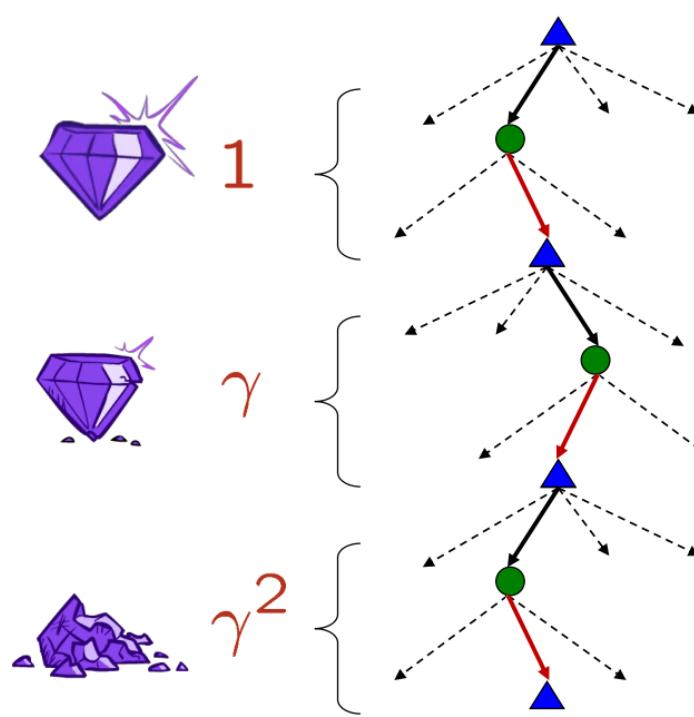
- Each time we descend a level, we multiply in the discount once

- Why discount?

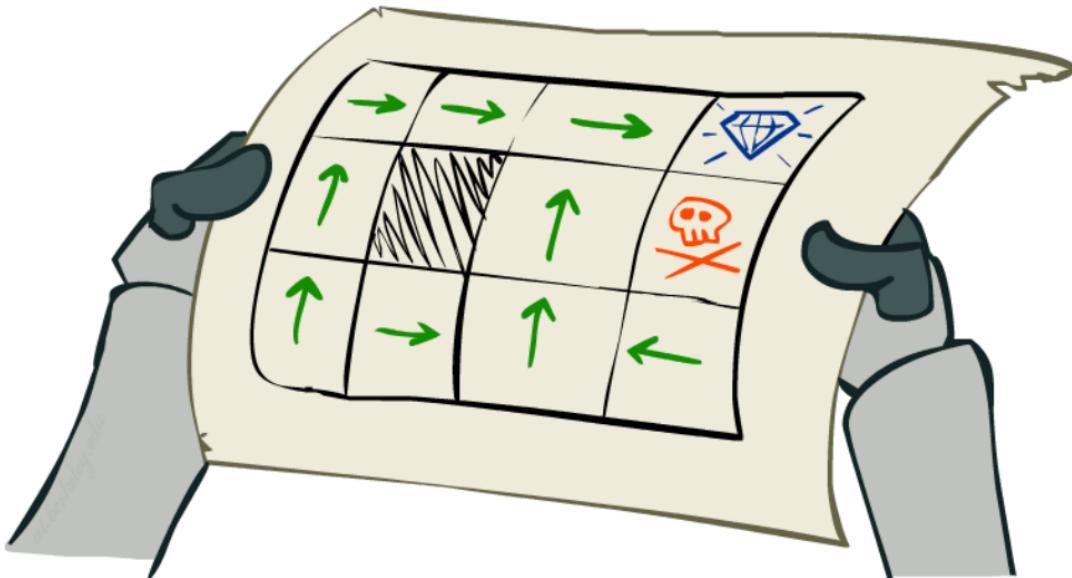
- Sooner rewards probably do have higher utility than later rewards
- Also helps our algorithms converge

- Example: discount of 0.5

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



# Solving MDPs



# Optimal Quantities

效用

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

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

$V(s)$  : start value     $V^*$  : best

- 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

此生之后

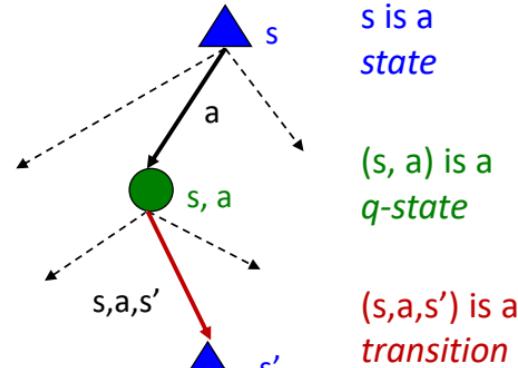
- The optimal policy:

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

$\text{Argmax } Q^* = \pi^*$



$Q$ -value



# may $Q^* = V^*$ Gridworld V Values

100 iter\$.

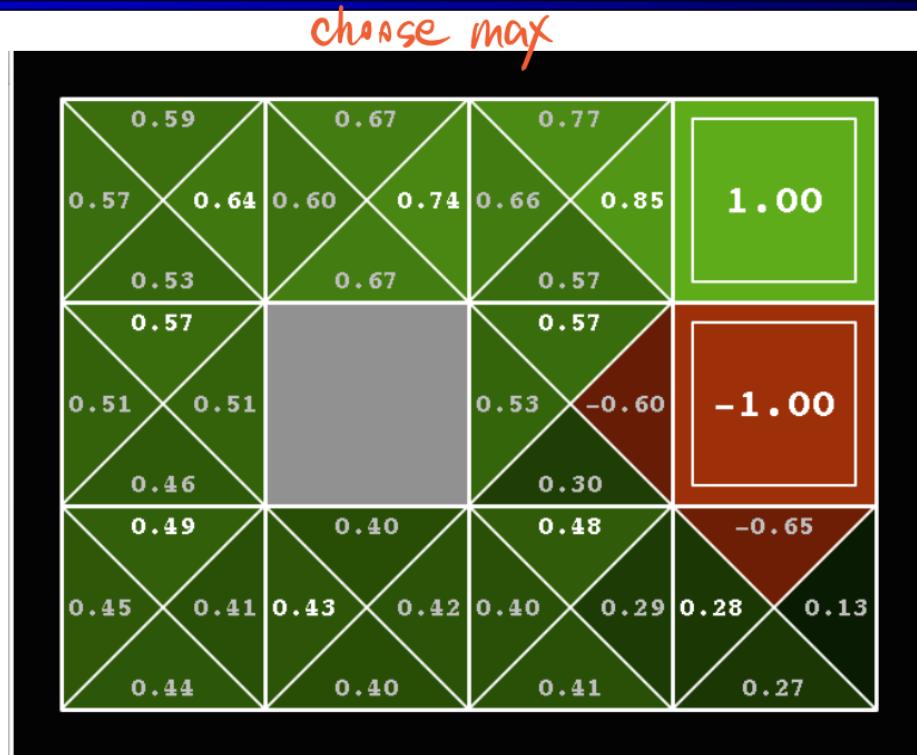


$\begin{matrix} 0.8 \\ \uparrow \\ 0.1 \end{matrix}$     $\begin{matrix} 0.1 \\ \rightarrow \\ 0.1 \end{matrix}$

Noise = 0.2  
Discount = 0.9  
Living reward = 0

✓  
no render

# Gridworld Q Values



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Values of States

- How to compute the value of a state
  - Expected utility under optimal action
  - This is just what expectimax computed!
- Recursive definition of value:

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

$P(<1!)$  present reward  $\gamma$  penalized future reward

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

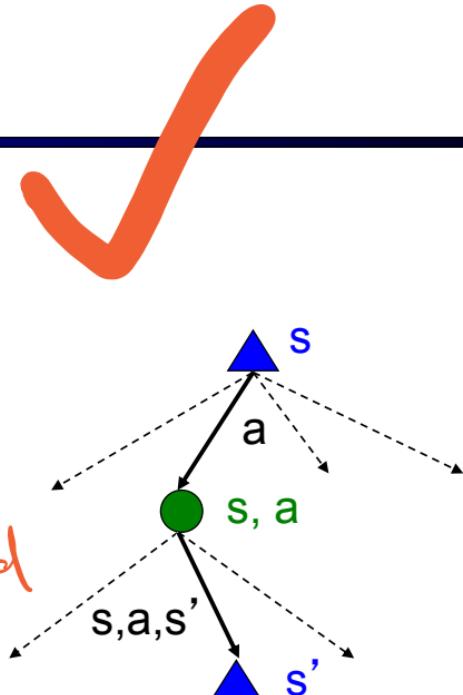
$$V^*(s) = \max_a \left( \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \right)$$

best action

The Bellman Equation

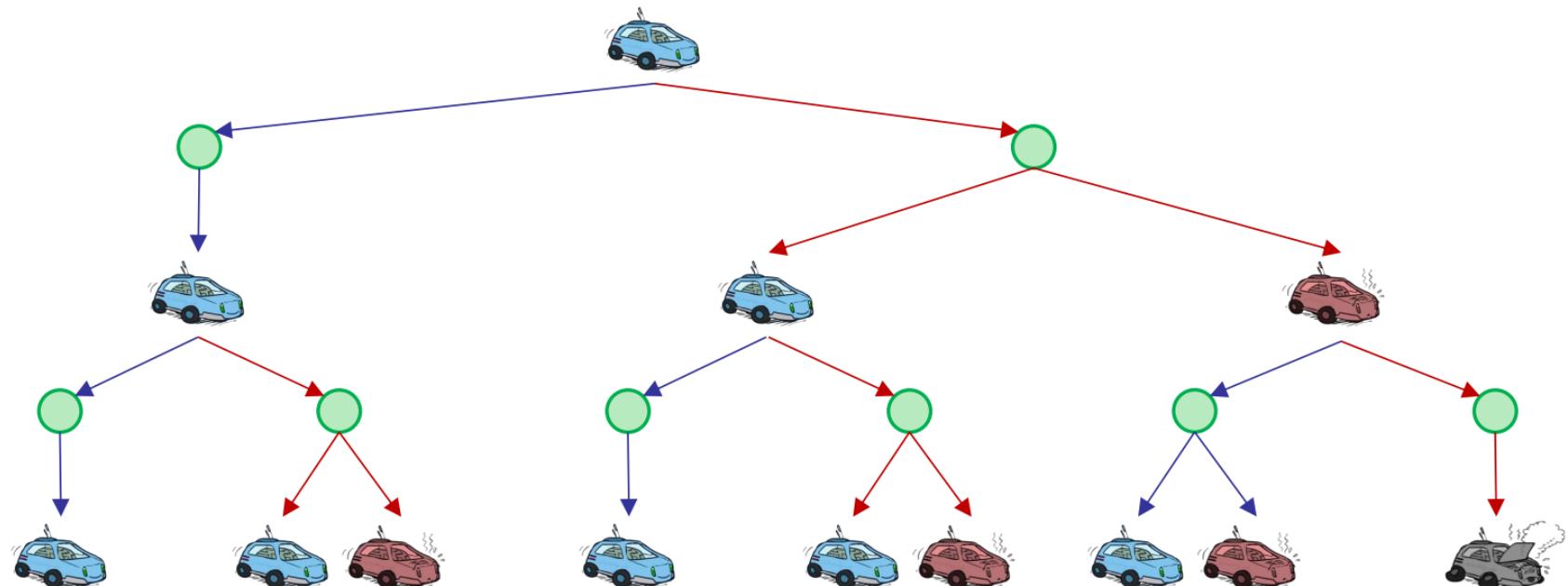
dp!

$s_k t_l$  (time left)

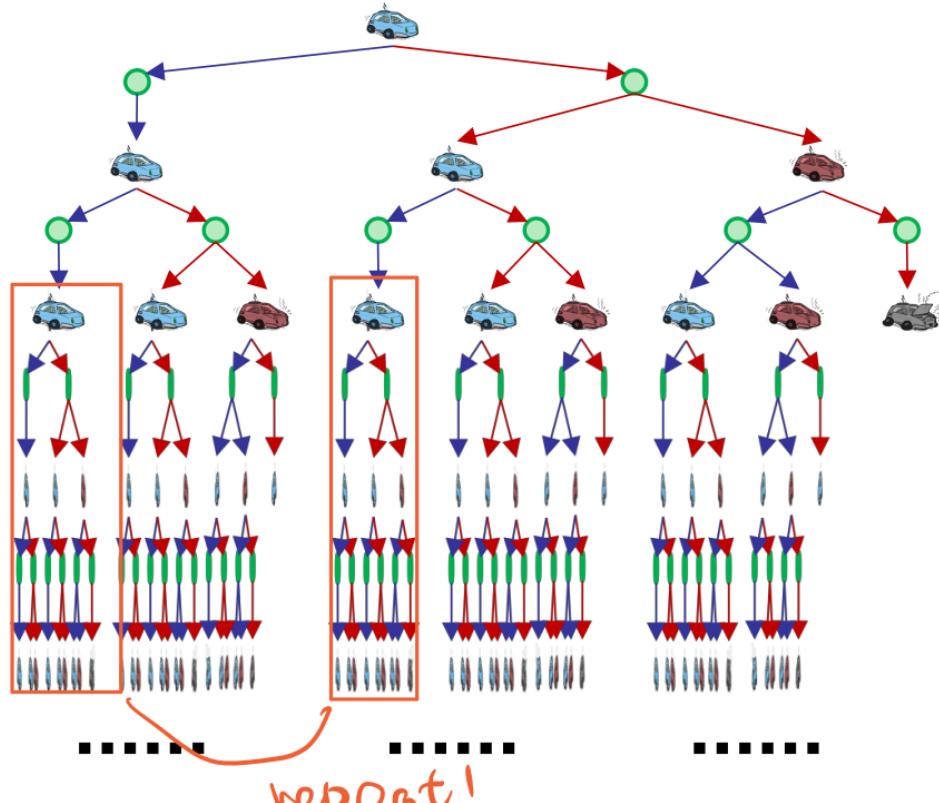


# Racing Search Tree

$S' : K$

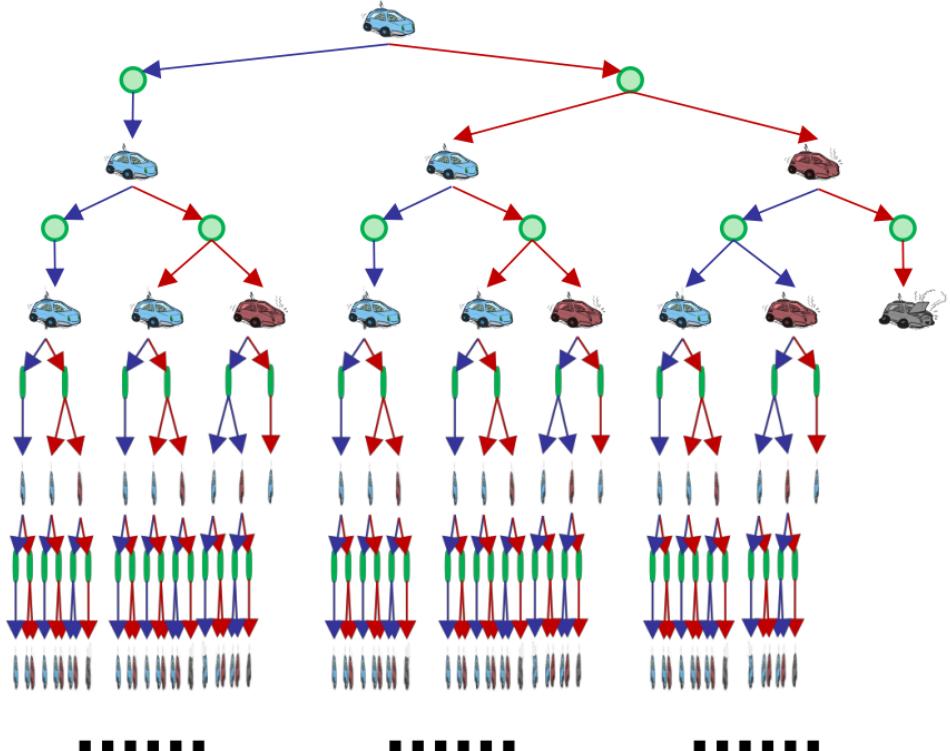


# Racing Search Tree



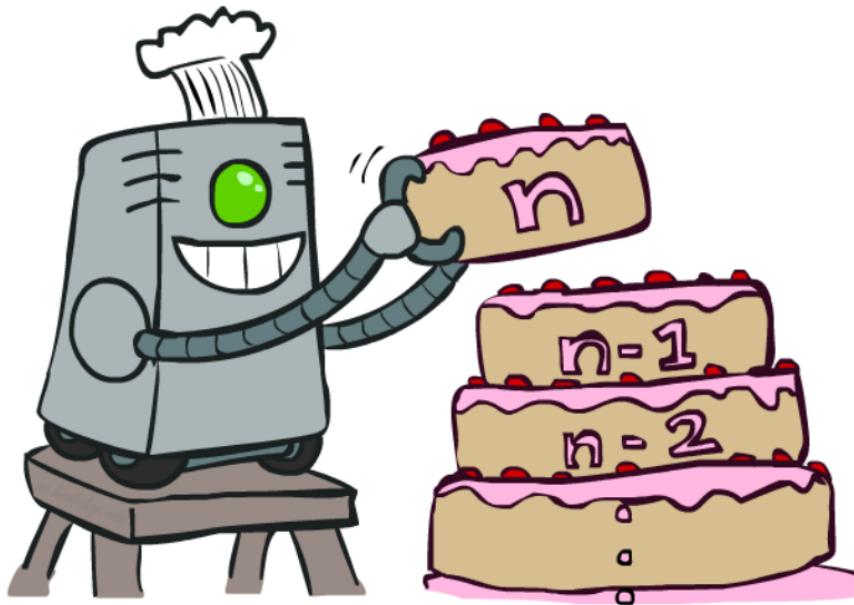
# Problems with Expectimax

- Problem 1: States are repeated
  - Idea: Only compute needed quantities once
- Problem 2: 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  $\gamma < 1$



# Value Iteration

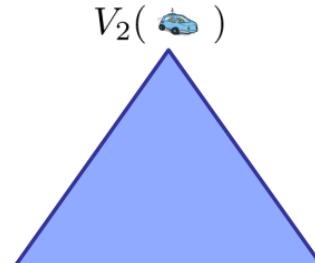
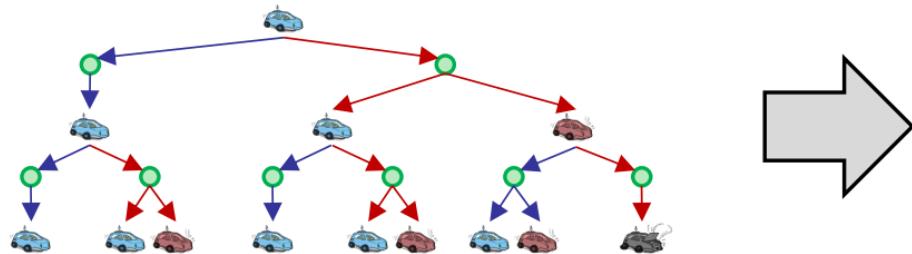
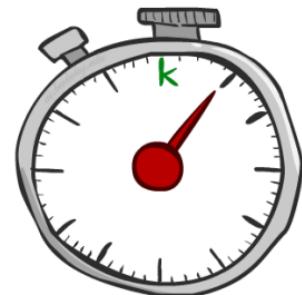
---



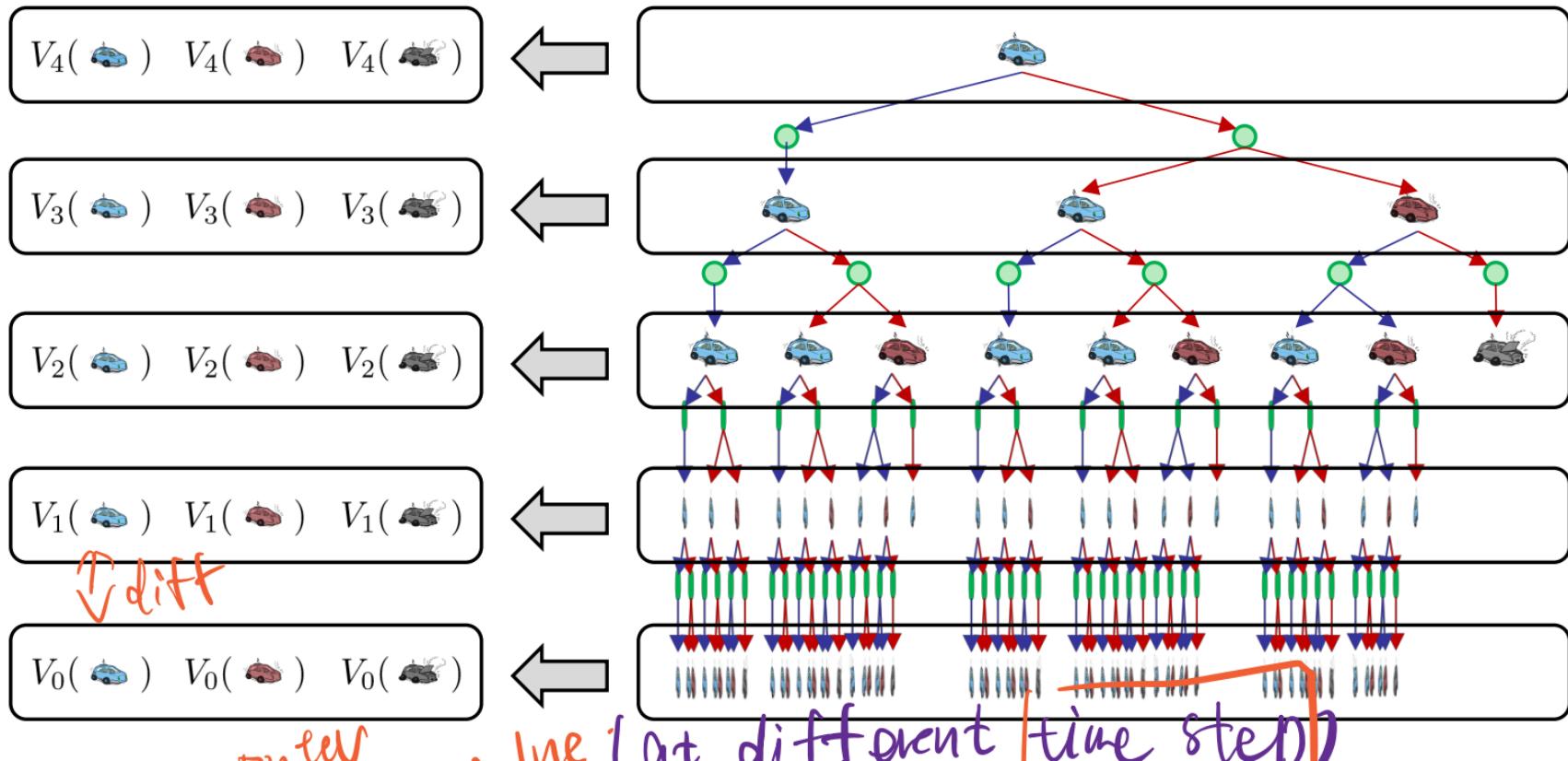
# Time-Limited Values

*k step limit*

- 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  $s$



# Computing Time-Limited Values



# Just compute 3 states's value function

## Value Iteration

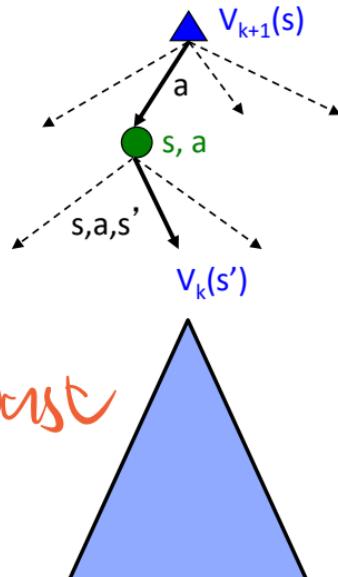
- Start with  $V_0(s) = 0$ : no time steps left means an expected reward sum of zero
- Given vector of  $V_k(s)$  values, do one ply of expectimax from each state:

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

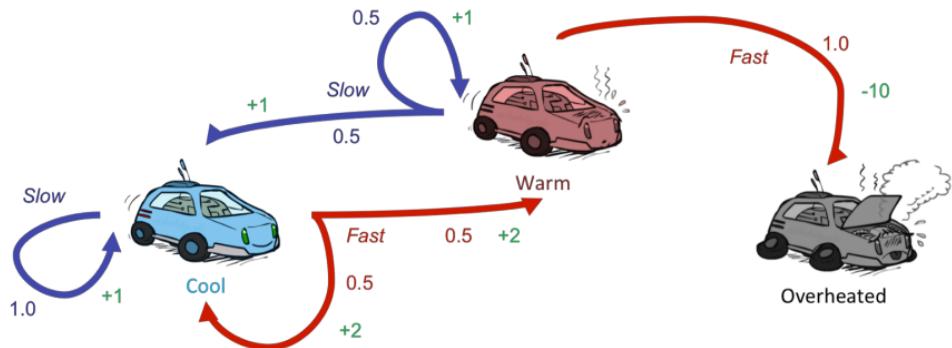
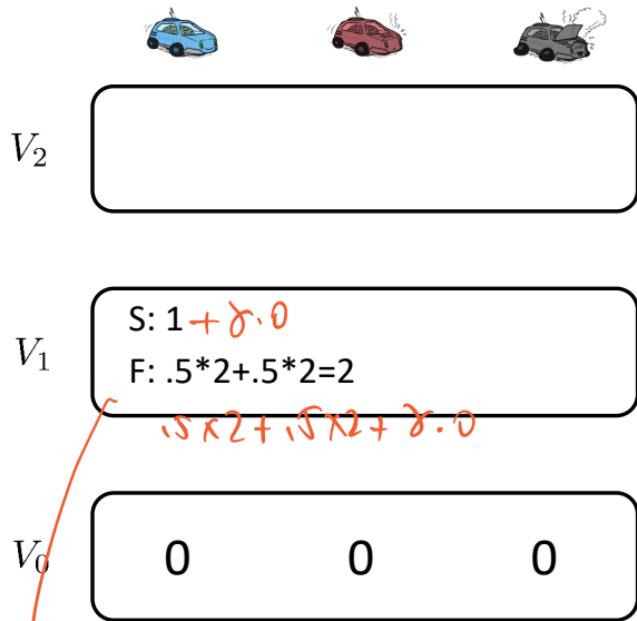
*time*

- Repeat until convergence
- Complexity of each iteration:  $O(S^2A)$
- Theorem: will converge to unique optimal values

bottom-up  
present  $\rightarrow$  past



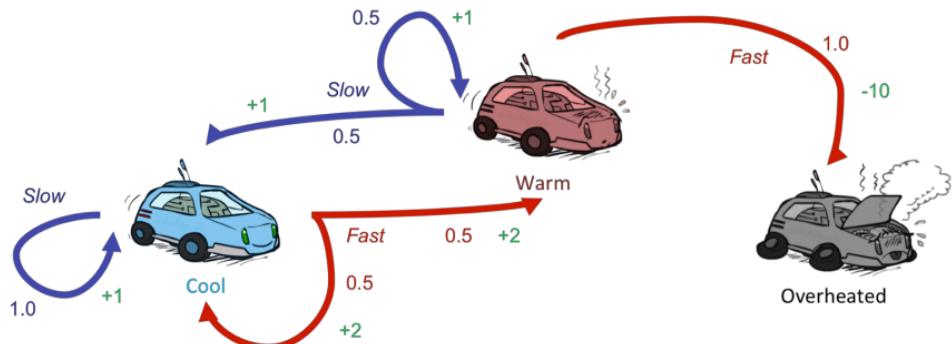
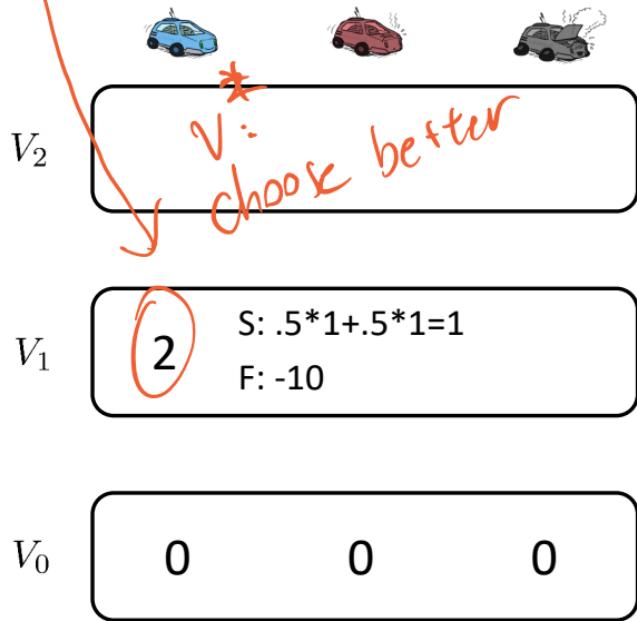
# Example: Value Iteration



Assume no discount!

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

# Example: Value Iteration

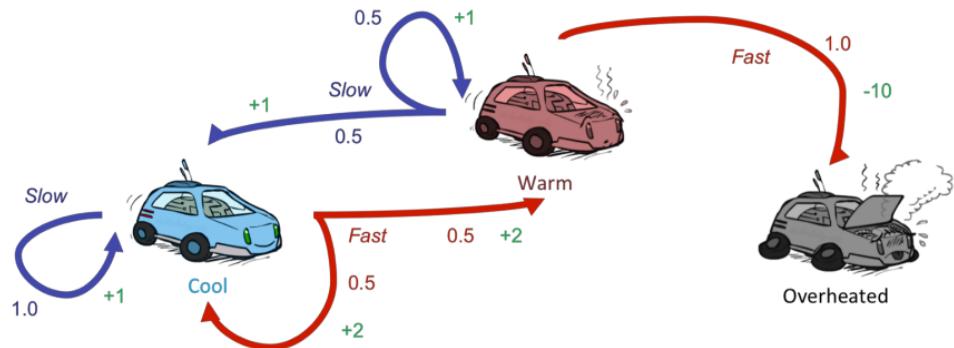


Assume no discount!

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

# Example: Value Iteration

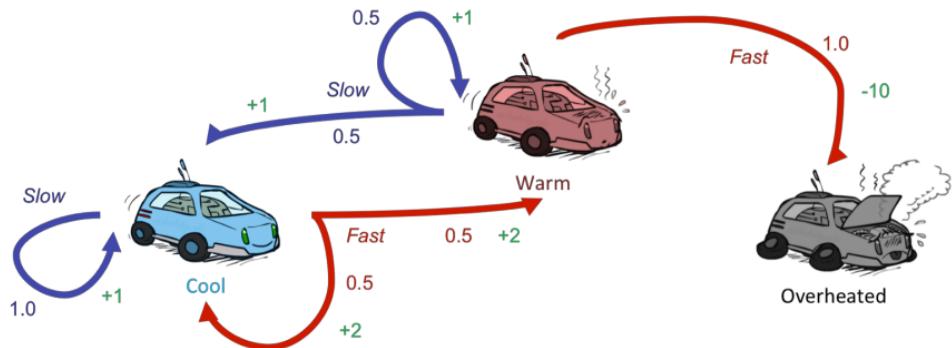
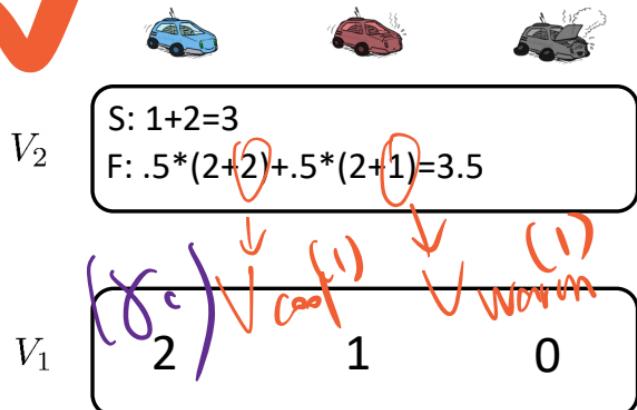
$V_2$			
$V_1$	2	1	0
$V_0$	0	0	0



Assume no discount!

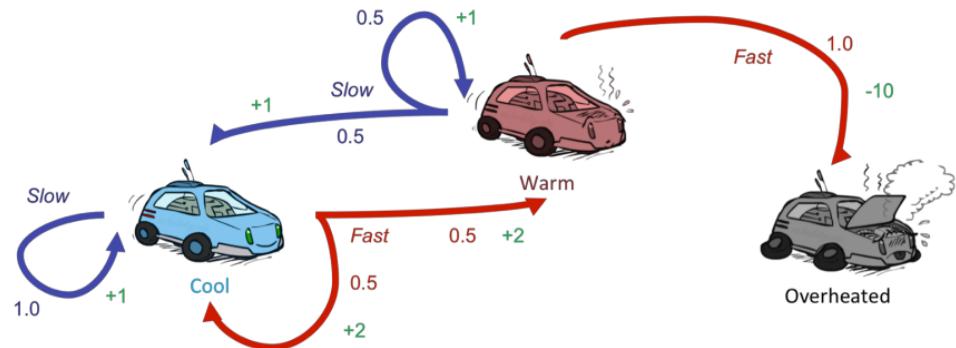
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

# Example: Value Iteration



# Example: Value Iteration

			
$V_2$	3.5 S: $.5*(2+1)+.5*(1+1)=2.5$ F: -10		
$V_1$	2      1      0		
$V_0$	0      0      0		

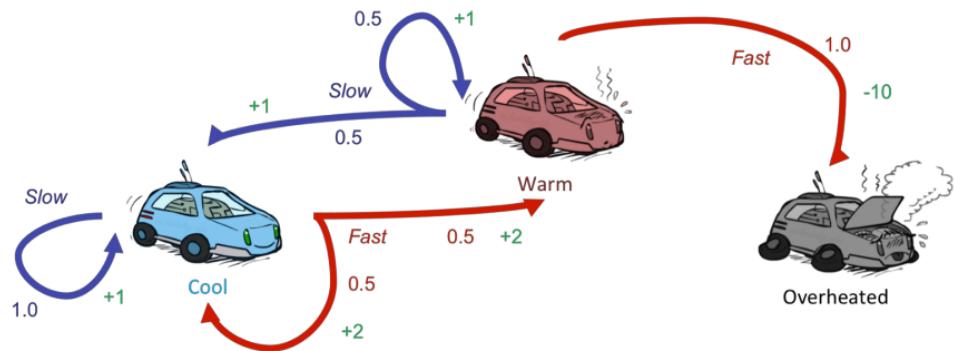


Assume no discount!

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

# Example: Value Iteration

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

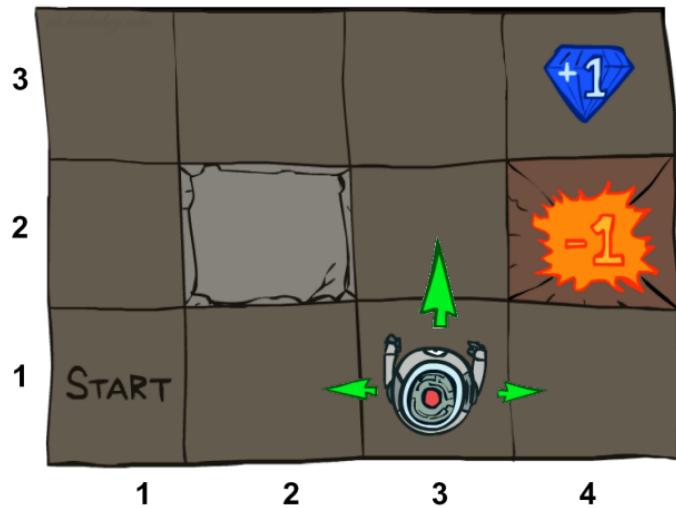


Assume no discount!

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

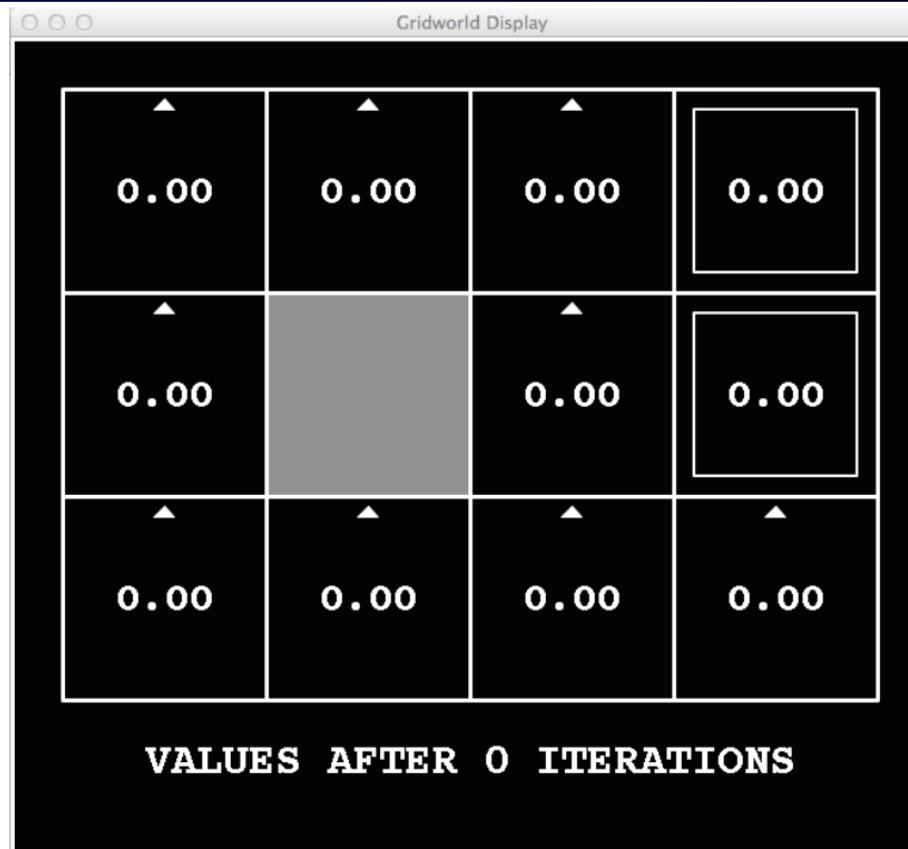
# Example: Grid World

- 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)



Suppose we get this reward by taking an “exit” action at a goal state

# $k=0$



Noise = 0.2  
Discount = 0.9  
Living reward = 0

$k=1$



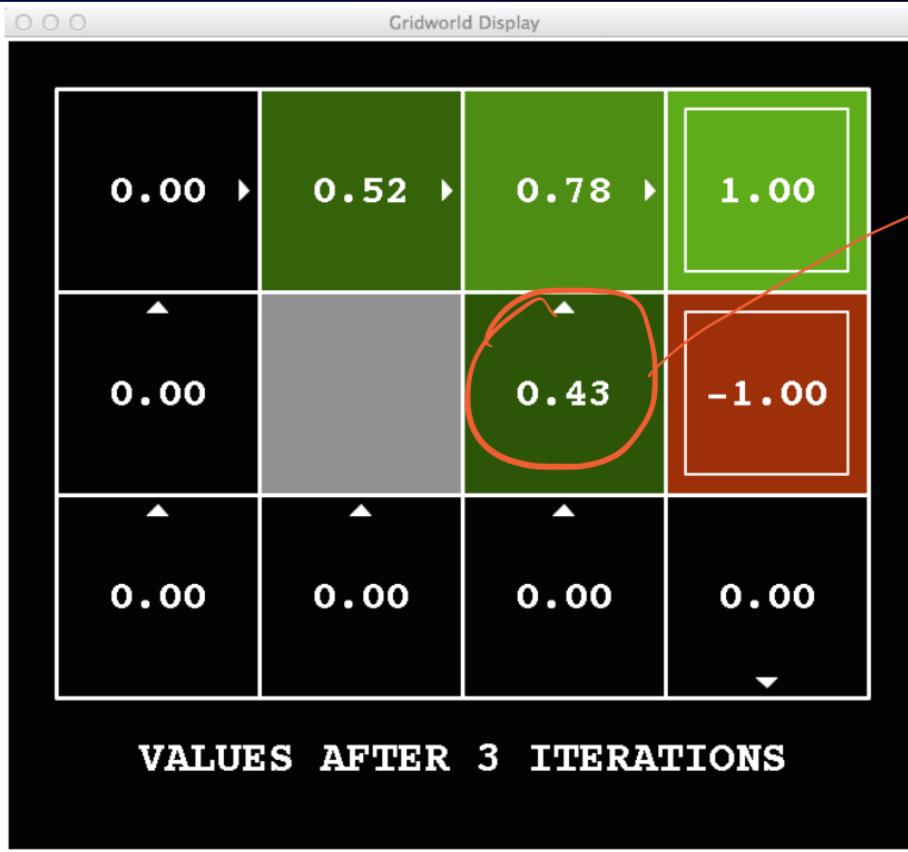
$k=2$   $80\%$   $0.8 \times 1.0 \times 0.9$  discount best



move

k=3

↑ 8%



$$\begin{array}{r} 0.78 \times 0.8 + x_0 \\ + 0 \times 0.9 \\ + -1 \times 0.9 \\ \hline \end{array}$$

~~$\rightarrow 0.10 \times 0.9 + 0.9$~~   
 $= 0.43.$

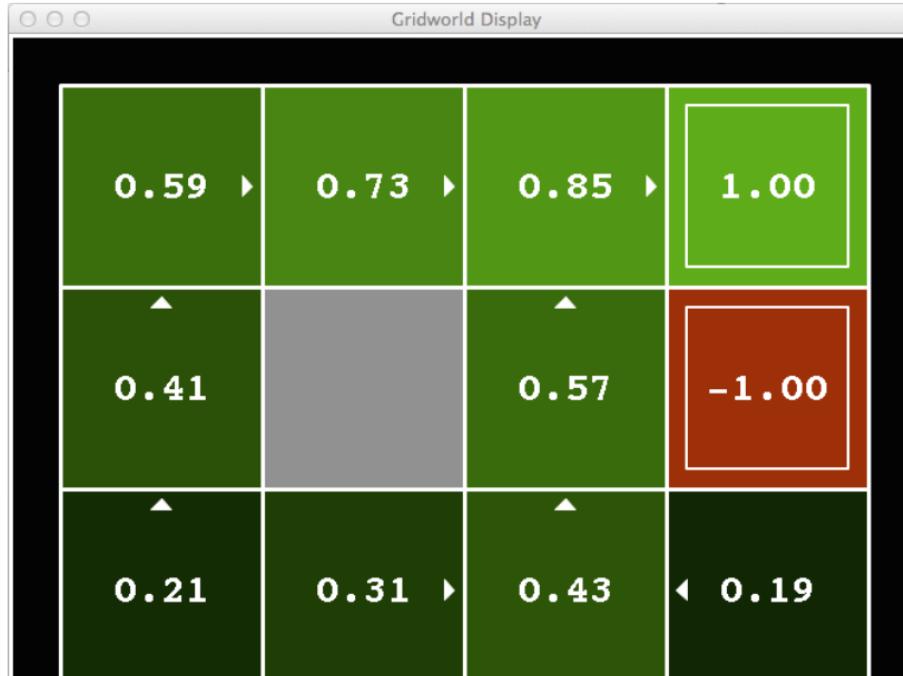
# k=4



# k=5



# k=6



VALUES AFTER 6 ITERATIONS

Noise = 0.2  
Discount = 0.9  
Living reward = 0

**k=7**



**VALUES AFTER 7 ITERATIONS**

Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=8



**k=9**



**VALUES AFTER 9 ITERATIONS**

Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=10



**k=11**



$k=12$



k=100

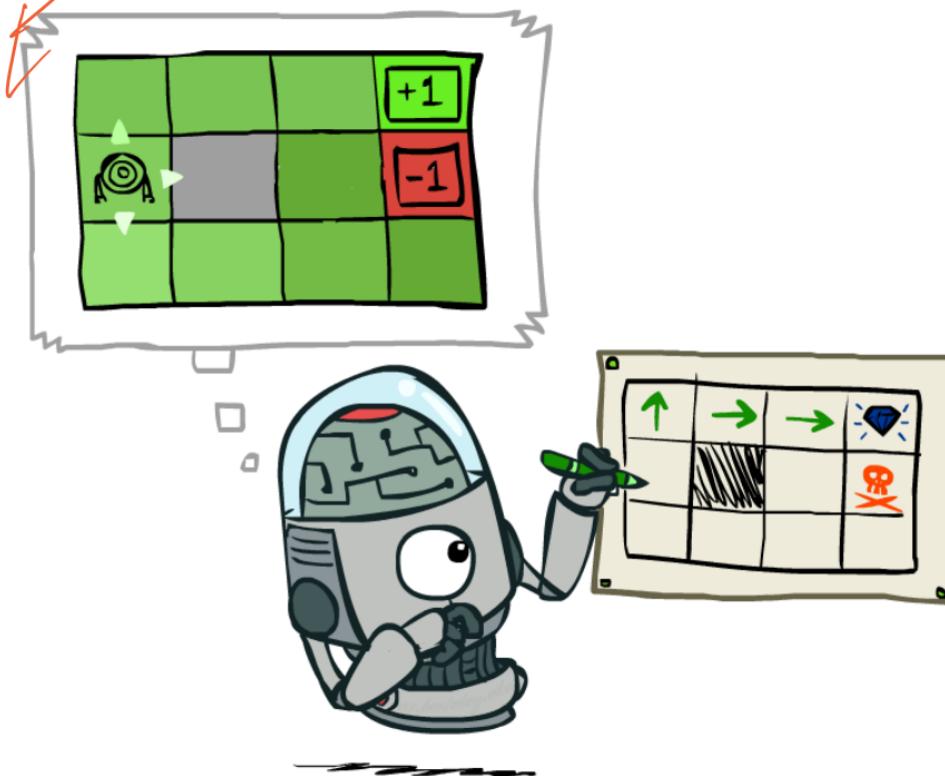
Value  
convergence



from value

## Policy Extraction

find  
action  
(policy)



# 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

e.g.:  $\leftarrow \text{Get } \pi^* \text{ for } 65\% \quad 0.89 \times 0.8 \times 0.9$

# Computing Actions from Q-Values

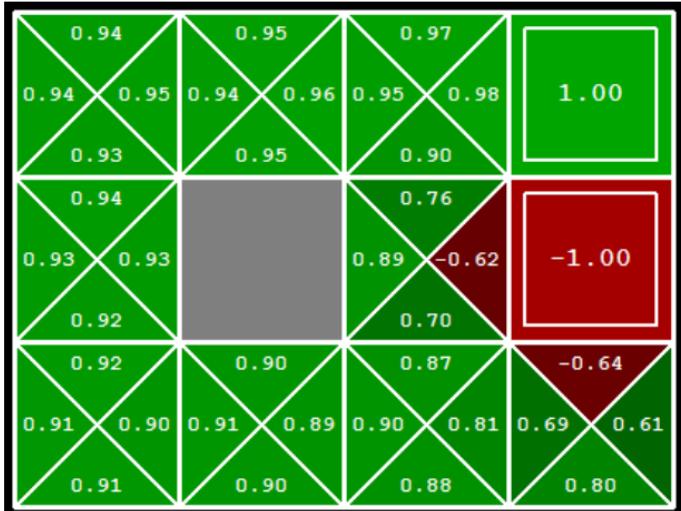
- Let's imagine we have the optimal q-values:

- How should we act?
  - Completely trivial to decide!

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

*Q values : Easier*

- Important: actions are easier to select from q-values than values!
- Q-values can also be computed in value iteration



# Q-Value Iteration

- Value iteration: find successive (depth-limited) values

- Start with  $V_0(s) = 0$
- Given  $V_k$ , calculate the depth  $k+1$  values for all states:

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

  
 $\underbrace{t+1}_{\tau}$

$\overline{\tau}$

- But Q-values are more useful, so compute them instead

- Start with  $Q_0(s, a) = 0$
- Given  $Q_k$ , calculate the depth  $k+1$  q-values for all q-states:

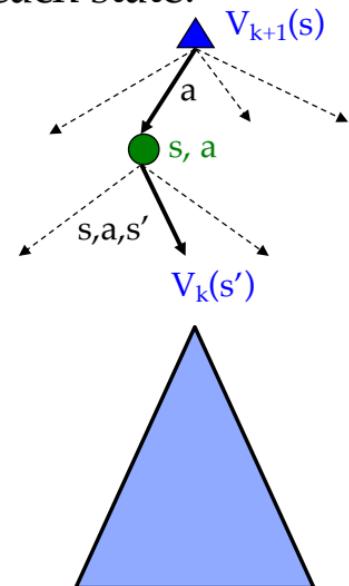
$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$$

# 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, 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



# Value Iteration

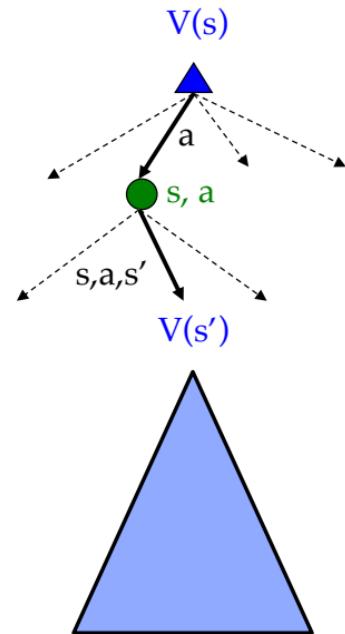
- Bellman equations **characterize** the optimal values:

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

- 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



# Value Iteration (again ☺ )

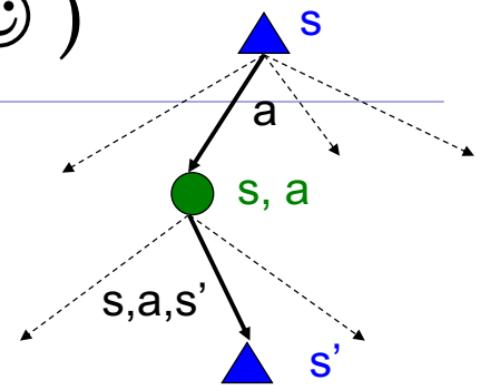
- Init:

$$\forall s: V(s) = 0$$

- Iterate:

$$\forall s: V_{new}(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

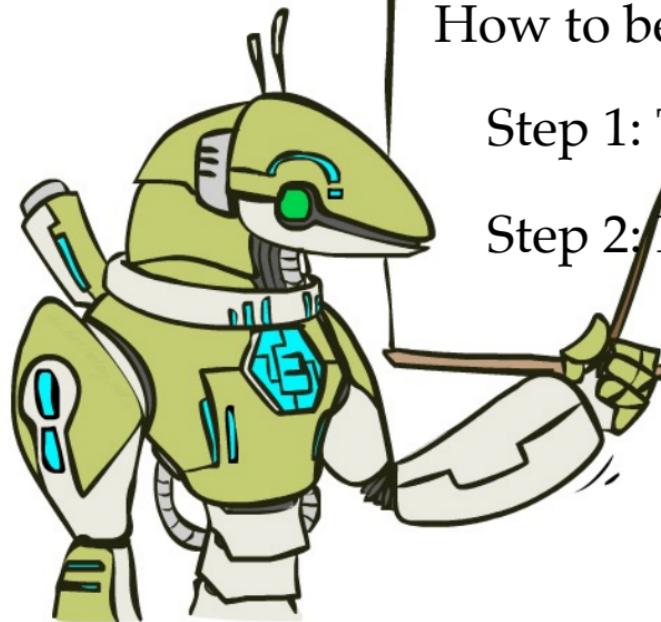
$$V = V_{new}$$



Note: can even directly assign to  $V(s)$ , which will not compute the sequence of  $V_k$  but will still converge to  $V^*$

# The Bellman Equations

= Dp.

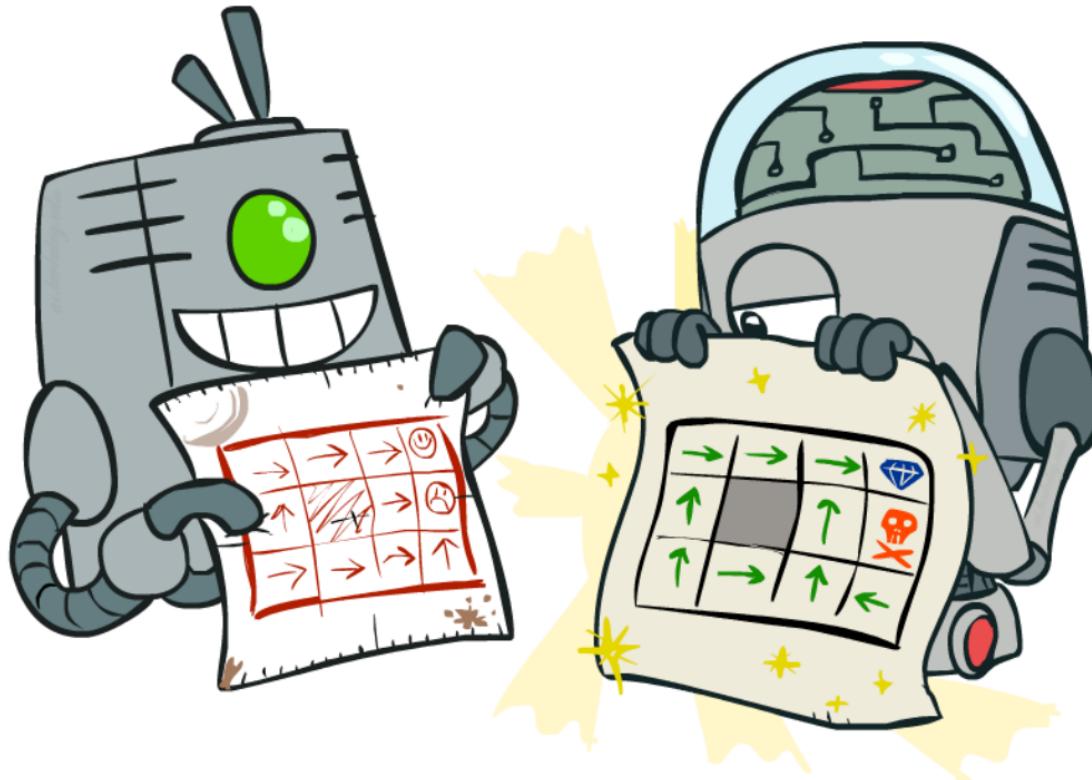


How to be optimal:

Step 1: Take correct first action

Step 2: Keep being optimal

# Policy Methods



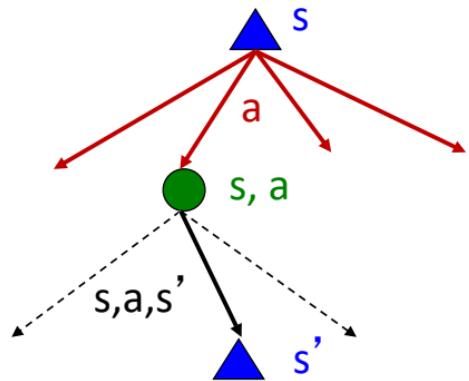
# Problems with Value Iteration

- 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
  - The policy often converges long before the values

policy faster than value



**k=12**



# k=100

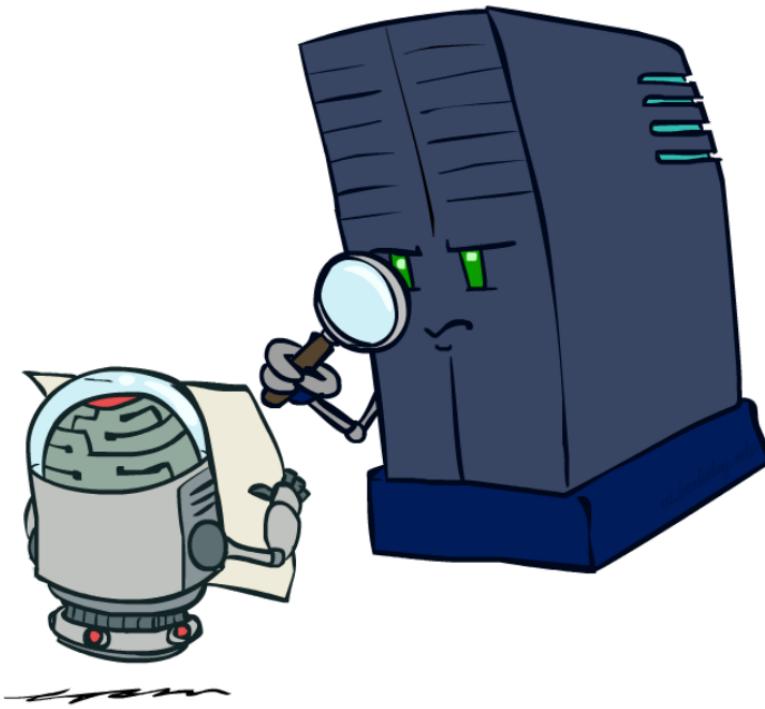


# Policy Iteration

- Policy iteration: an alternative approach for value iteration
    - Step 1: Policy evaluation: calculate utilities for some fixed (not optimal) policy
    - 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
  - It's still optimal!
  - Can converge (much) faster under some conditions
- Calculate state's value under policy*
- value*

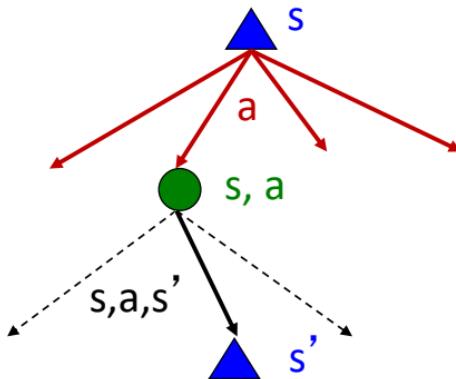
# Step 1: Policy Evaluation

---

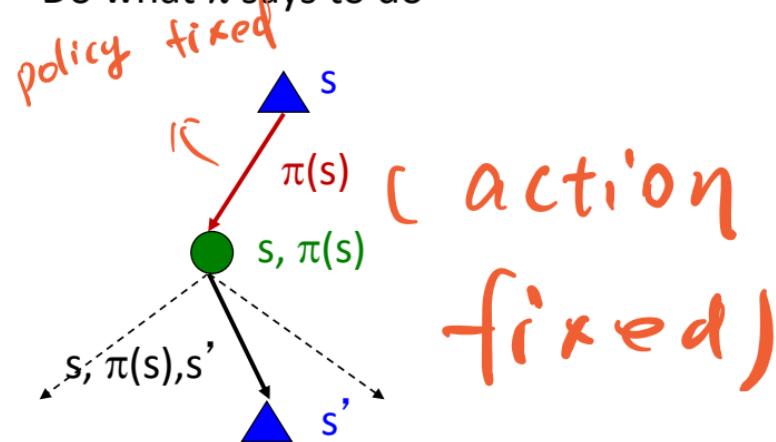


# Fixed Policies

Do the optimal action



Do what  $\pi$  says to do



- 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

# Utilities for a Fixed Policy

- The utility of a state  $s$ , under a fixed policy  $\pi$ :

$V^\pi(s)$  = expected utility starting in  $s$  and following  $\pi$

given policy  $\pi$

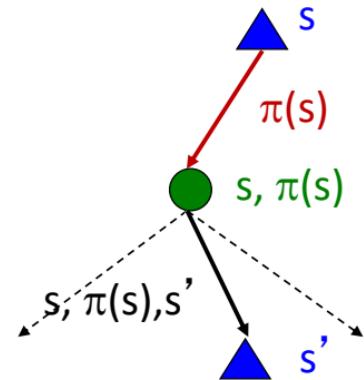
- Recursive relation (one-step look-ahead):

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



(transition func)

Bellman func



# Policy Evaluation

- How do we calculate the values under a fixed policy  $\pi$ ?

- Idea 1: Iterative updates (like value iteration)

- Start with  $V_0^\pi(s) = 0$
- Given  $V_k^\pi$ , calculate the depth  $k+1$  values for all states:

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

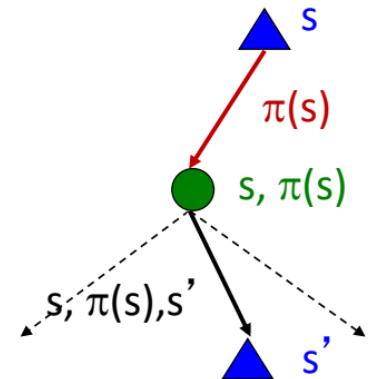
- Repeat until convergence
- Efficiency:  $O(S^2)$  per iteration

~~$O(S^2 \lambda)$~~

- Idea 2: Without the maxes, the Bellman equations are just a linear system

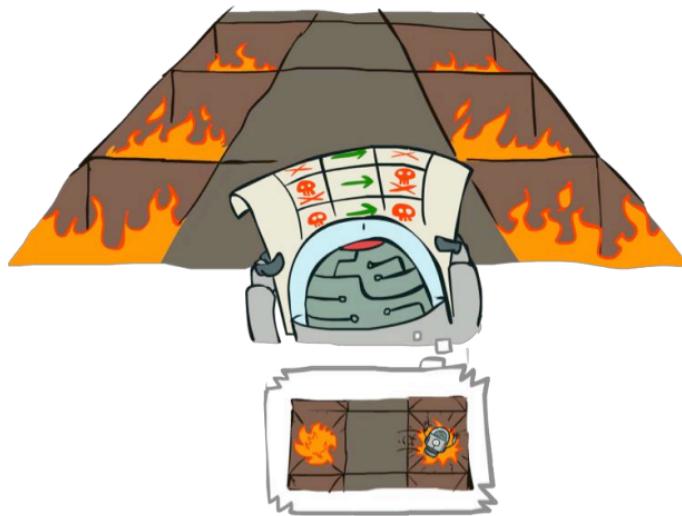
$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

- Solvable with a **linear system solver**



# Example: Policy Evaluation

Policy: Always Go Right



-10.00	100.00	-10.00
-10.00	1.09	-10.00
-10.00	-7.88	-10.00
-10.00	-8.69	-10.00

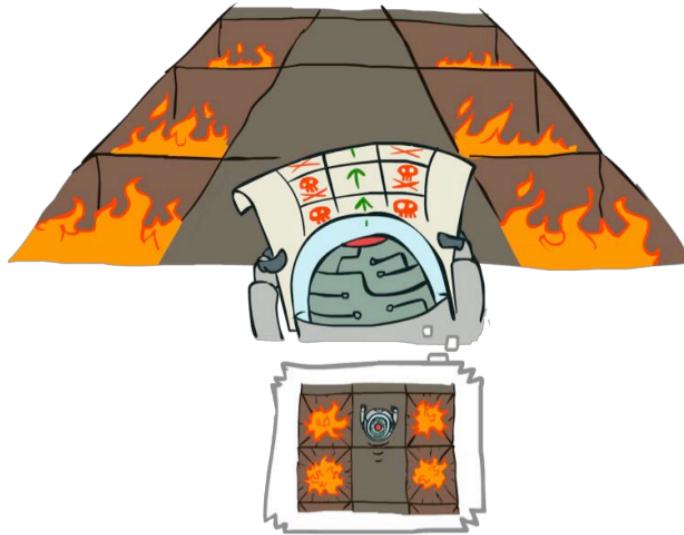
$$\begin{aligned} & \alpha | \\ & \uparrow \\ & V_1 \rightarrow 0.8 \\ & \downarrow \\ & 0.1 \times 0.8 \times \gamma_0 \\ & \times \gamma \end{aligned}$$

$$V_1 = 0.1 \times 100 \times 0.8 + 0.1 \times V_2 \times 0.8$$

$$\begin{aligned} V_1 &= f(V_2) \\ V_2 &= f(V_1, V_3) \\ V_3 &= f(V_2) \end{aligned}$$

# Example: Policy Evaluation

Always Go Forward



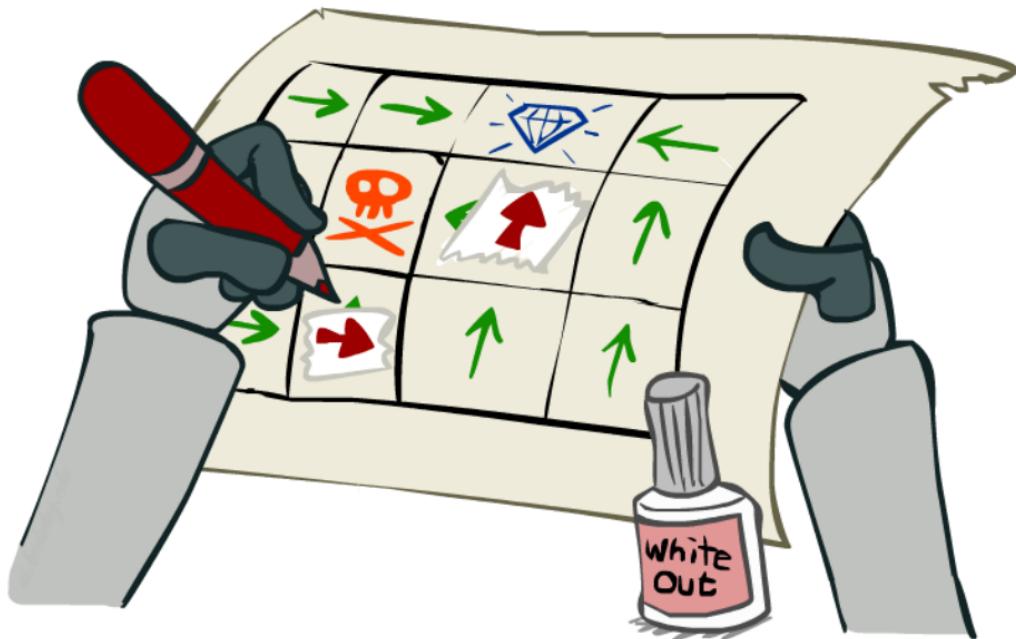
-10.00	100.00	-10.00
-10.00	70.20	-10.00
-10.00	48.74	-10.00
-10.00	33.30	-10.00

The table illustrates the results of policy evaluation for a 4x4 grid world. The values are as follows:

- Top row: -10.00 (red), 100.00 (green), -10.00 (red)
- Second row: -10.00 (red), 70.20 (green), -10.00 (red)
- Third row: -10.00 (red), 48.74 (green), -10.00 (red)
- Bottom row: -10.00 (red), 33.30 (green), -10.00 (red)

Arrows between the green cells indicate the value function is converging towards 100.00.

## Step 2: Policy Improvement



Actions:

# Value $\Rightarrow$ Policy Policy Improvement

- Step 2: Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

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

Just like value iter / expect max.

value of previous round

- Policy Iteration: repeat the two steps until policy converges

$\downarrow$

$V(\max)$

$\downarrow$

$R(s, a, s')$

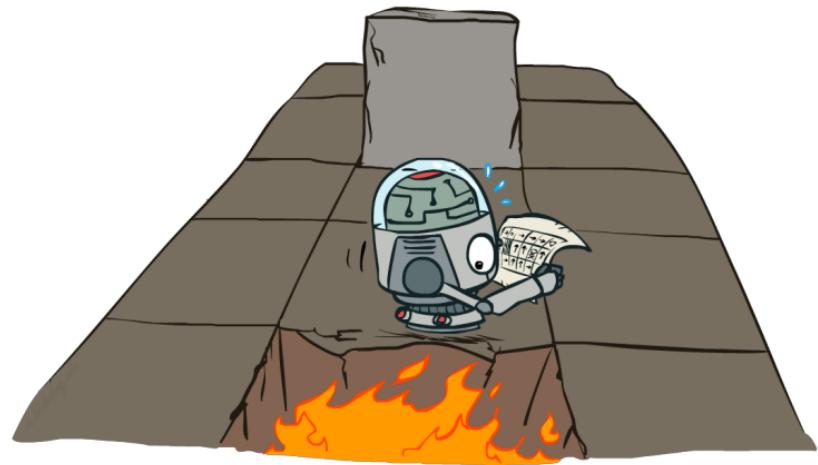
from Policy

# Value Iteration vs. Policy Iteration

- 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)
  - May converge faster
- Both are dynamic programs for solving MDPs

# Summary

- Markov Decision Process
  - States S, Actions A, Transitions  $P(s' | s, a)$ , Rewards  $R(s, a, s')$
- Quantities:
  - Policy, Utility, Values, Q-Values
- Solve MDP
  - Value iteration
  - Policy iteration



# 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

just compute values.