

# CSE 573: Artificial Intelligence

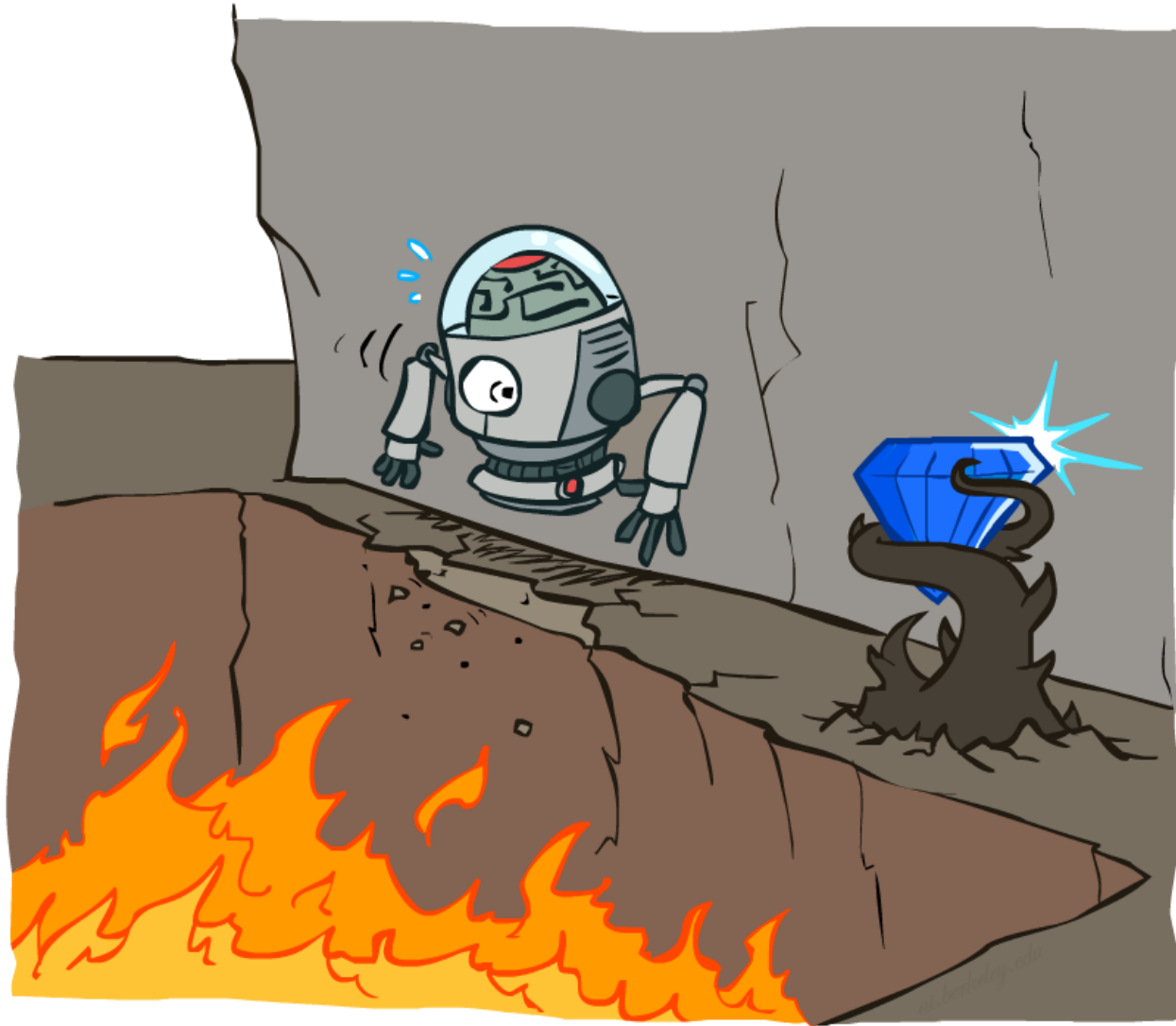
## Markov Decision Processes



slides adapted from  
Stuart Russel, Dan Klein, Pieter Abbeel from [ai.berkeley.edu](http://ai.berkeley.edu)  
And Hanna Hajishirzi, Jared Moore, Dan Weld

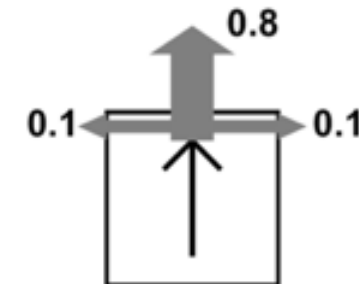
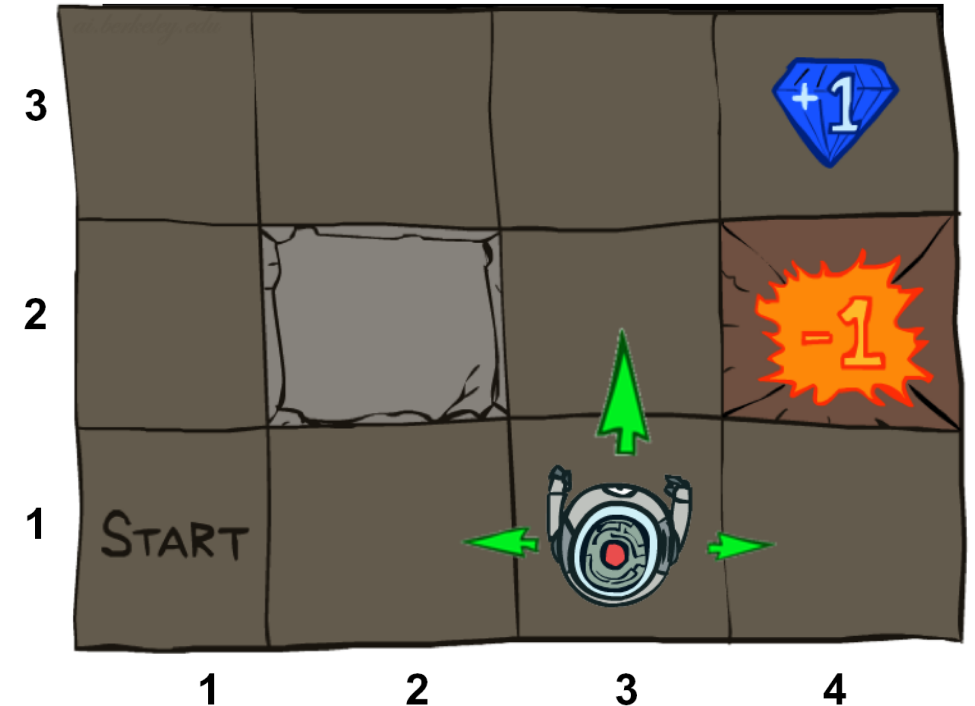
# Non-Deterministic Search

---



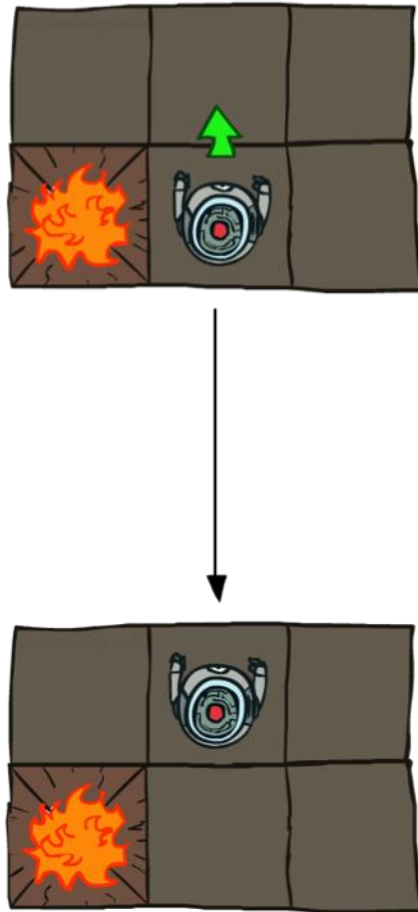
# Example: Grid World

- 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

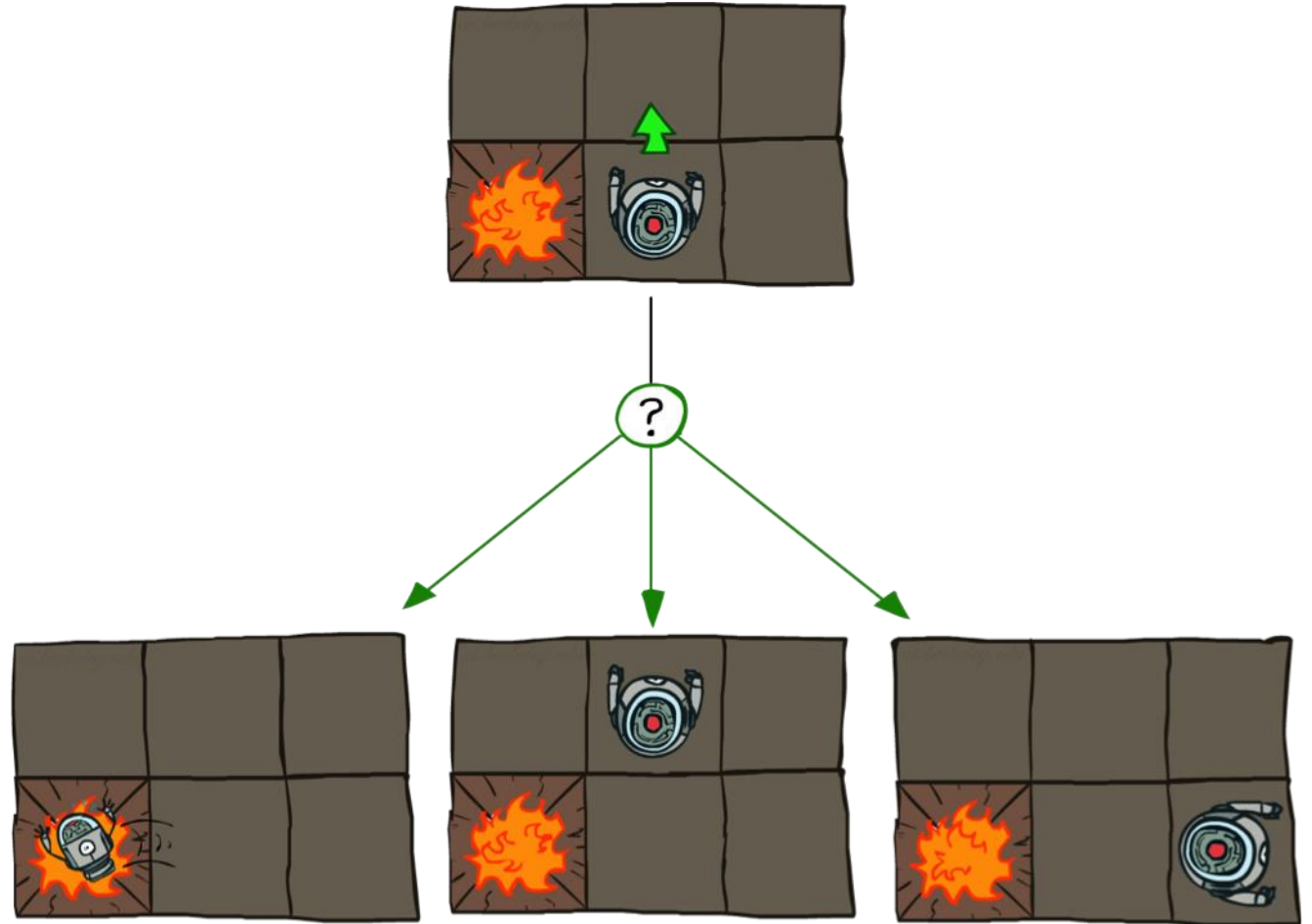


# Grid World Actions

Deterministic Grid World



Stochastic Grid World



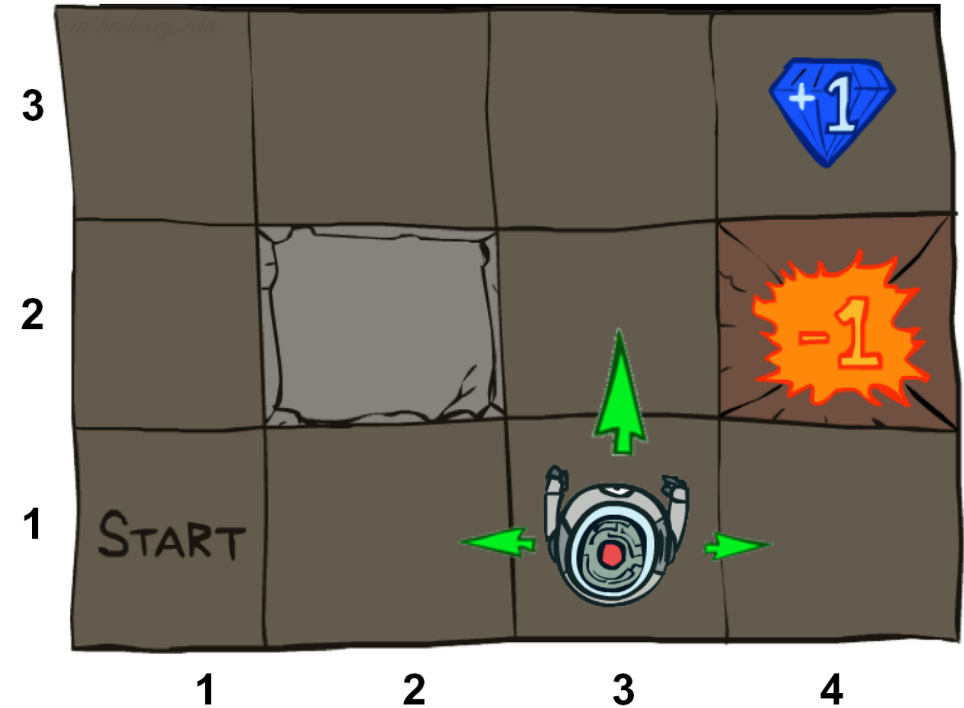
# Markov Decision Processes

- 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

$T(s_{11}, E, s_{11}) = 1$   
 $\dots$   
 $T(s_{31}, N, s_{11}) = 0$   
 $\dots$   
 $T(s_{31}, N, s_{32}) = 0.8$   
 $T(s_{31}, N, s_{21}) = 0.1$   
 $T(s_{31}, N, s_{41}) = 0.1$   
 $\dots$

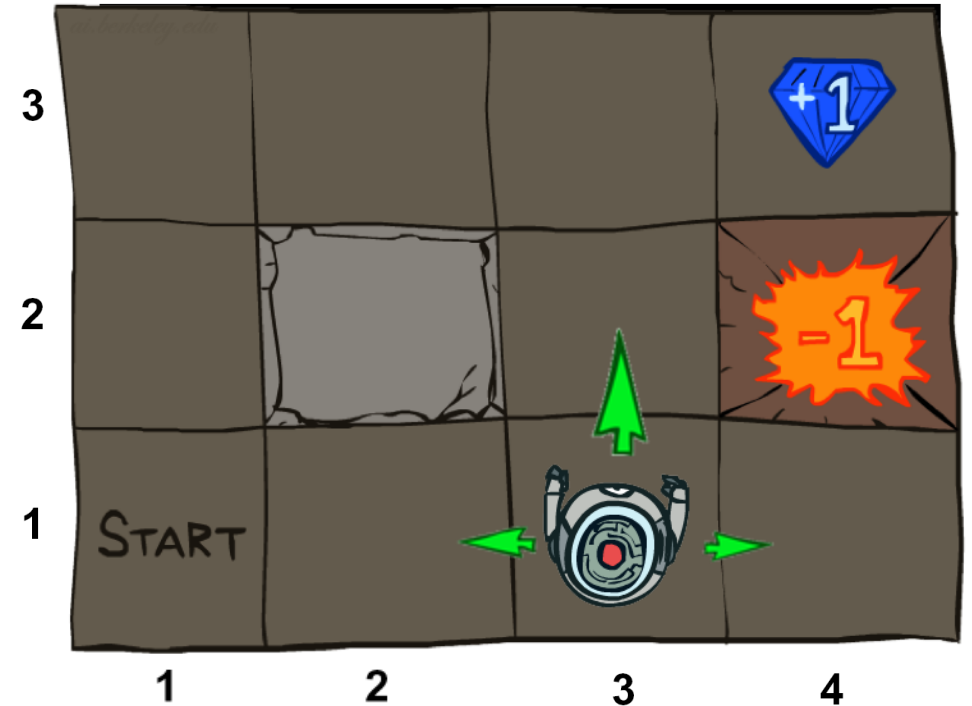
This is a Big Table!  
 $11 \times 11 \times 11 = 1331$  entries

For now, we give this as input to the agent



# 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  $R(s')$



$R(s_{32}, N, s_{33}) = 0.01$   
 $R(s_{32}, N, s_{42}) = 1.01$   
 $R(s_{33}, E, s_{43}) = 0.99$

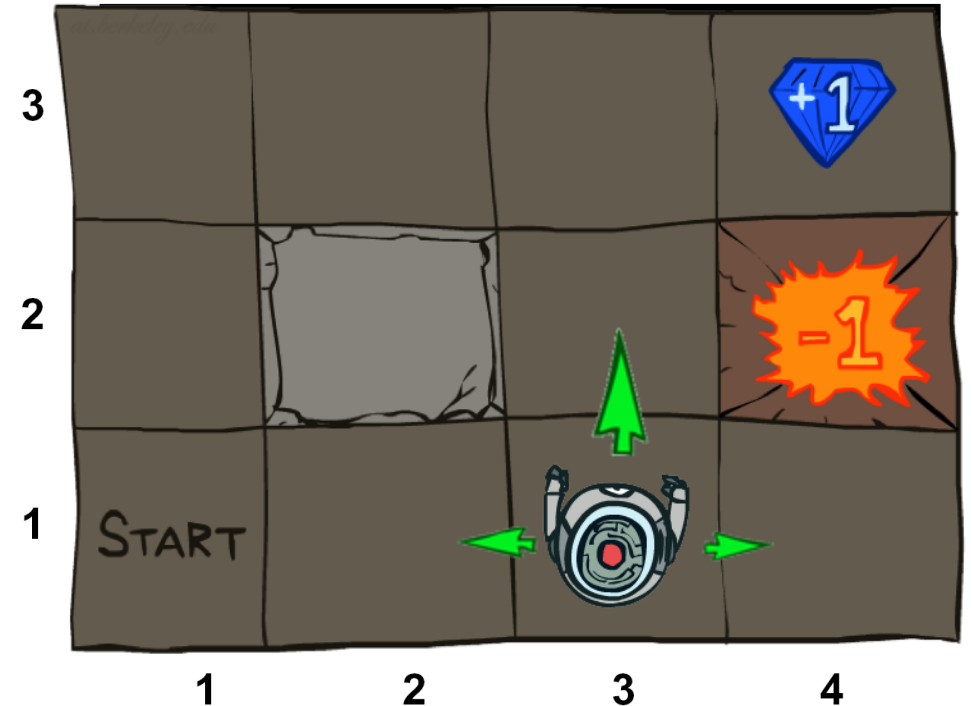
*Cost of breathing*

R is also a Big Table!

For now, we also give this to the agent

# 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

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

=

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

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

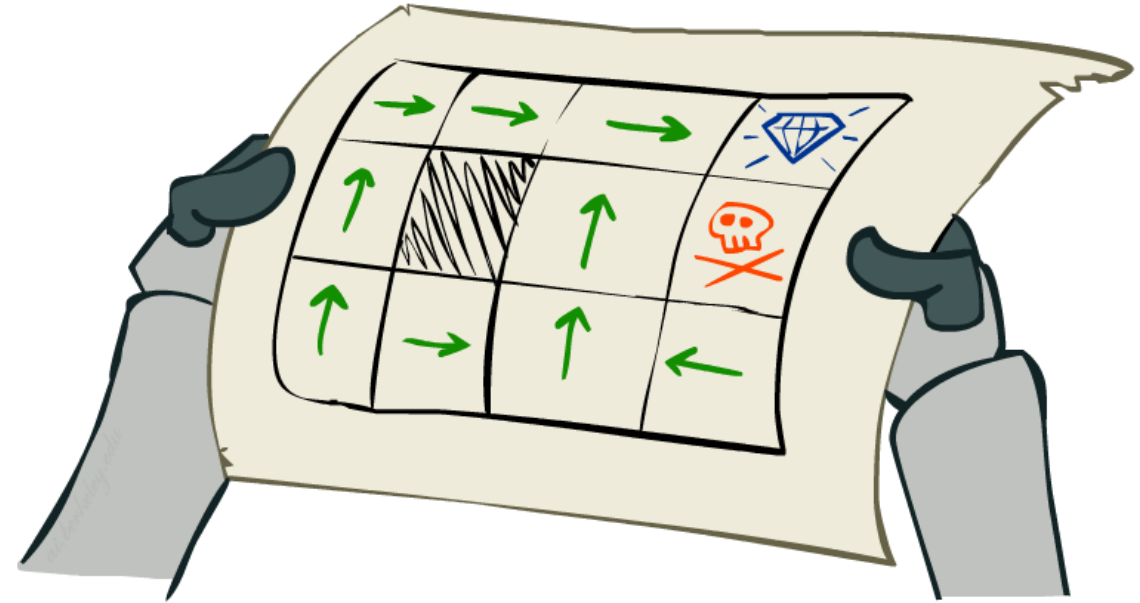


Andrey Markov  
(1856-1922)



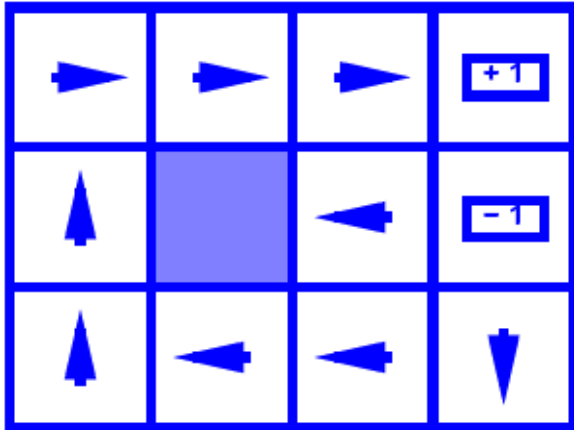
# Policies

- 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

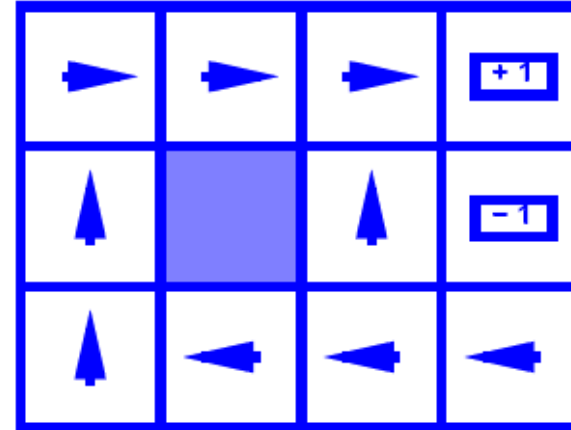


Optimal policy when  $R(s, a, s') = -0.4$  for all non-terminals  $s$

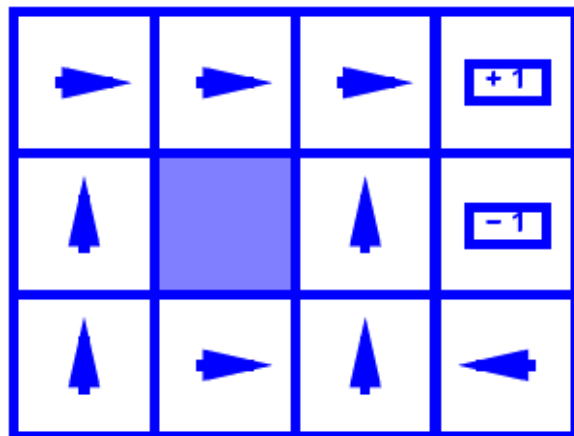
# Optimal Policies



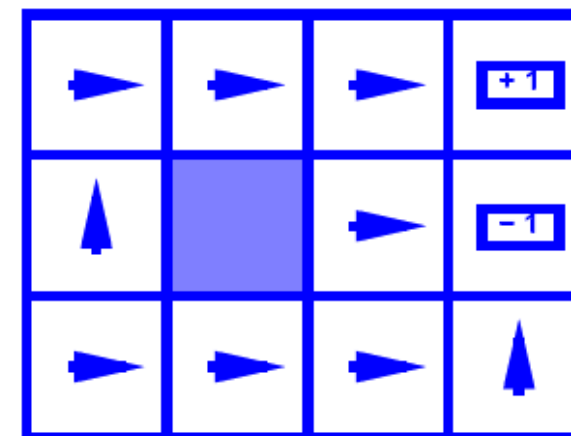
$R(s) = -0.01$



$R(s) = -0.03$



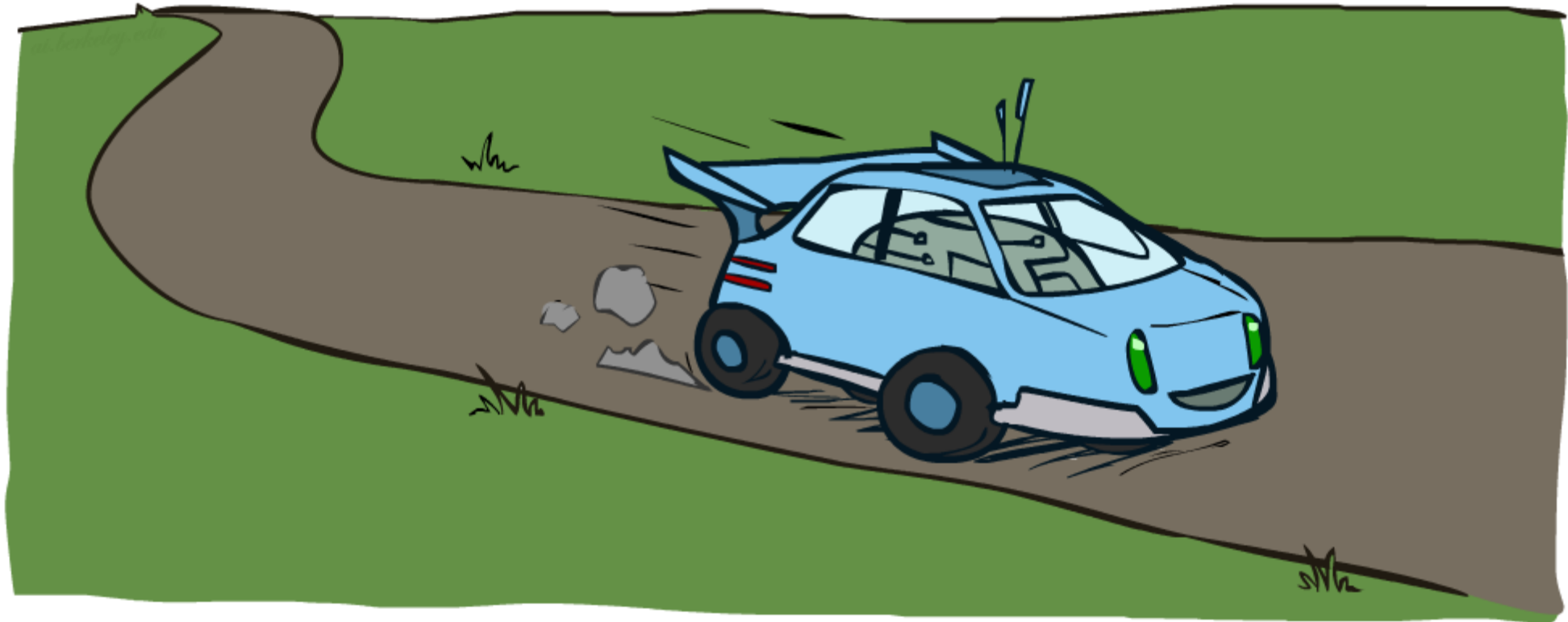
$R(s) = -0.4$



$R(s) = -2.0$

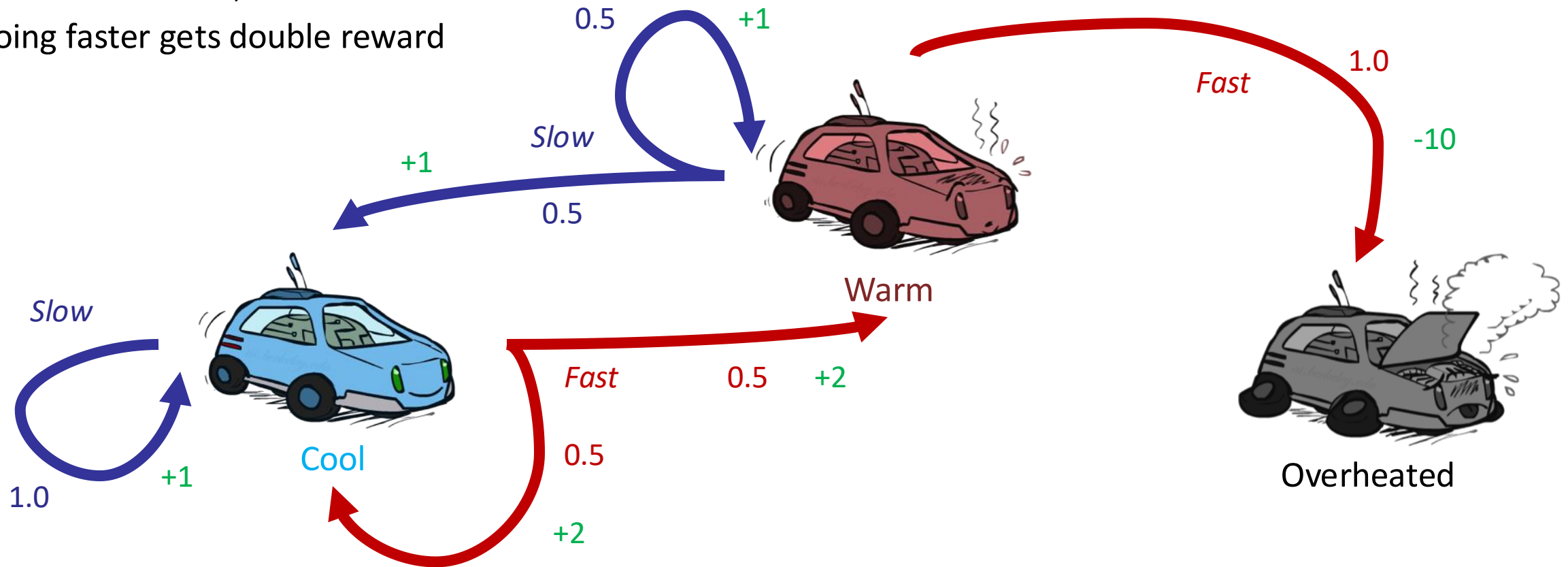
# Example: Racing

---



# Example: Racing

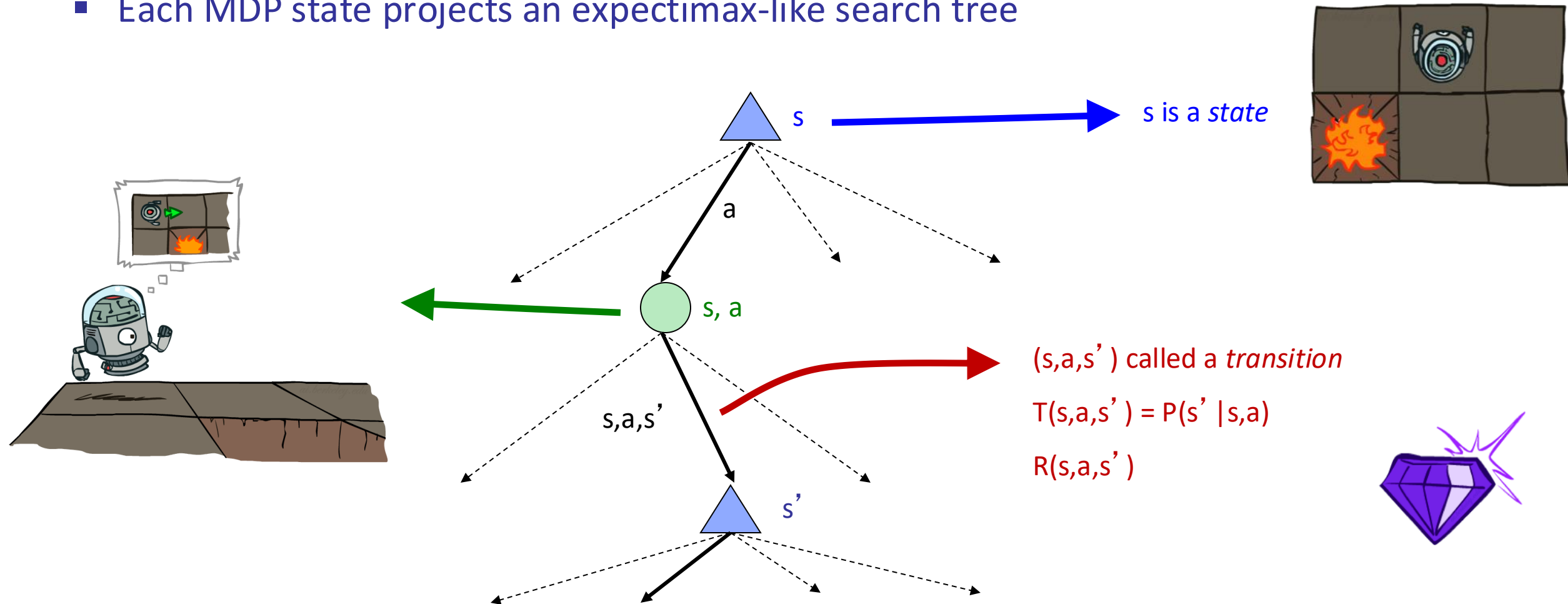
- A robot car wants to travel far, quickly
- Three states: **Cool**, **Warm**, Overheated
- Two actions: **Slow**, **Fast**
- Going faster gets double reward



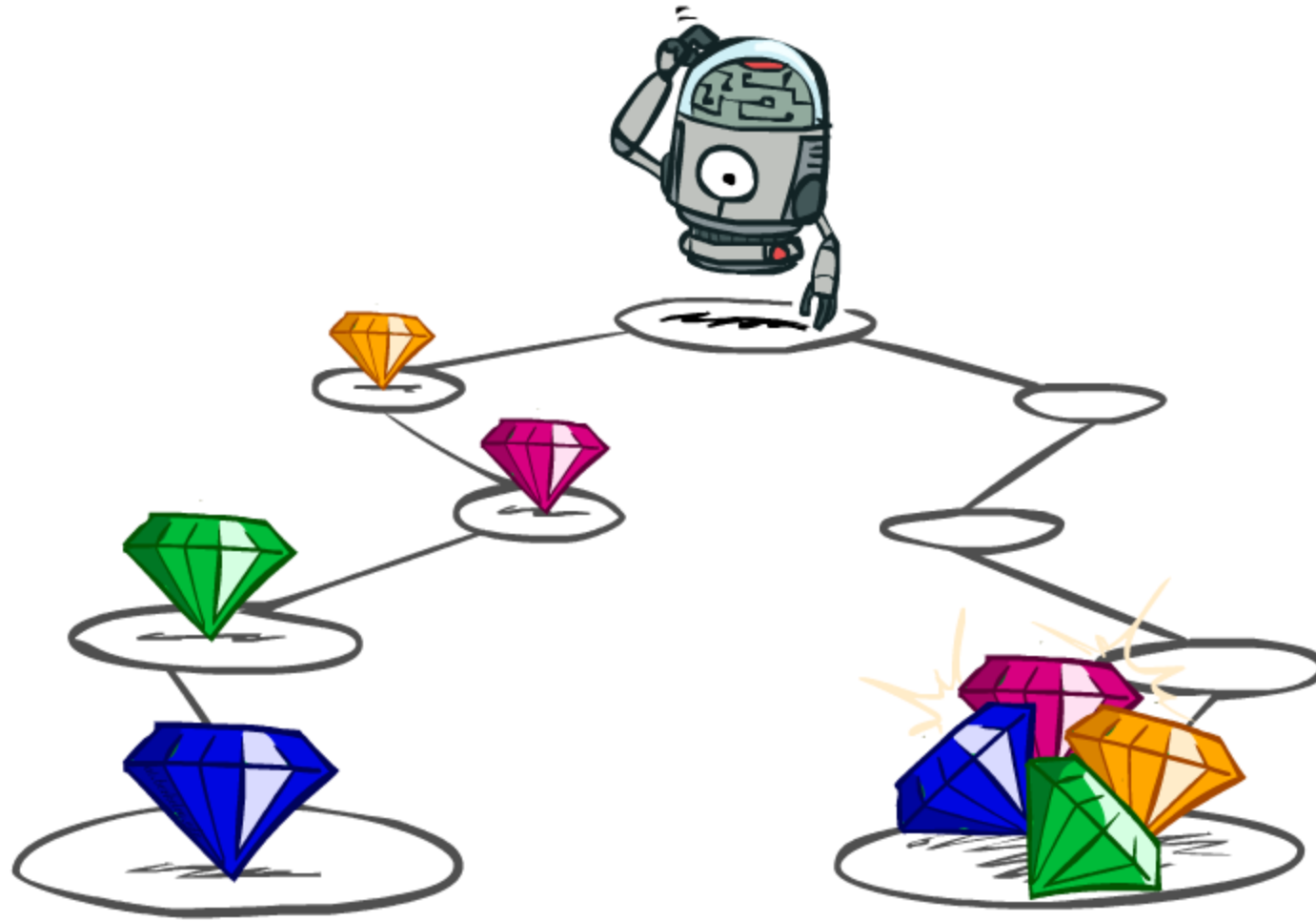


# MDP Search Trees

- Each MDP state projects an expectimax-like search tree

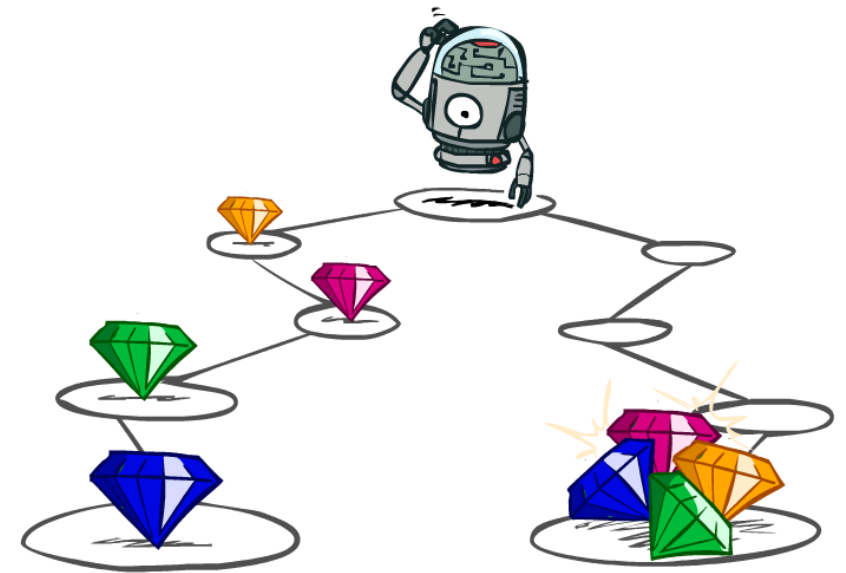


# Utilities of Sequences



# 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

# Discounting

- How to discount?

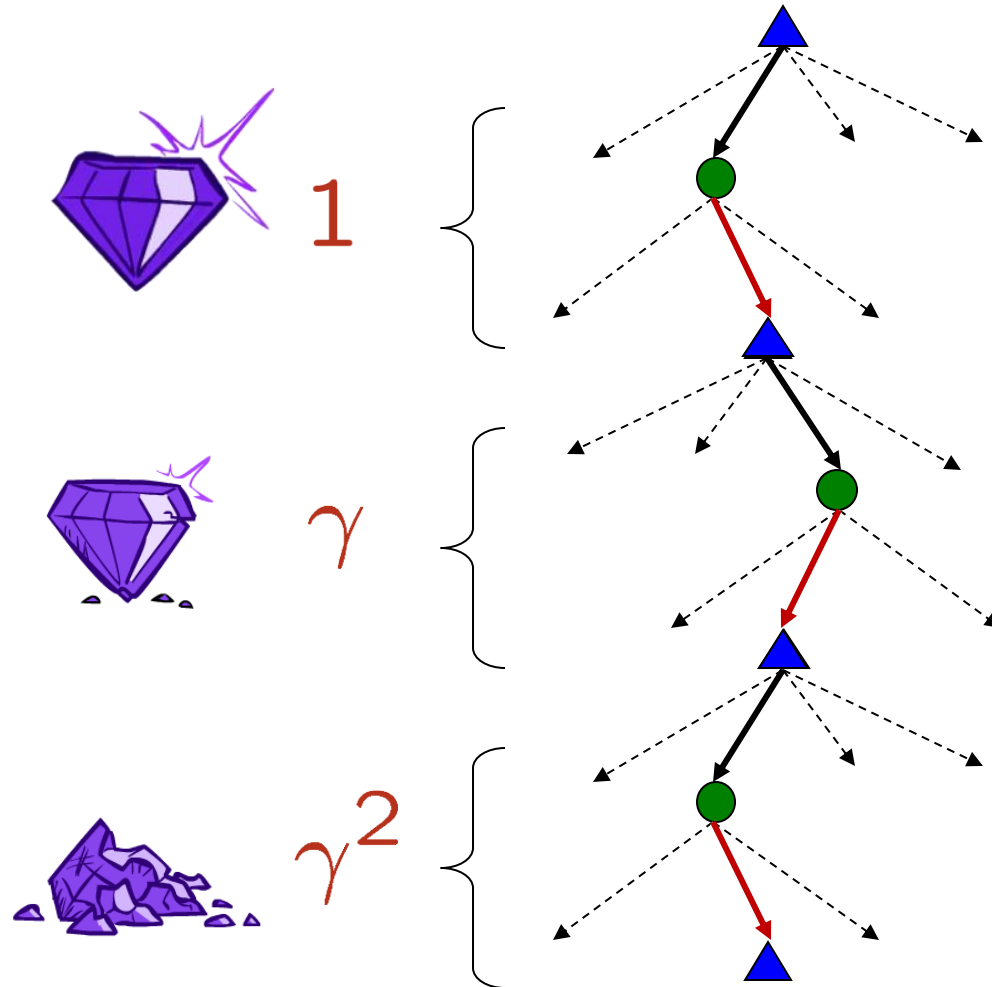
- Each time we descend a level, we multiply in the discount once

- Why discount?

- Think of it as a gamma chance of ending the process at every step
- Also helps our algorithms converge

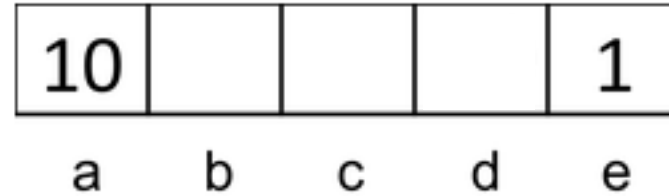
- Example: discount of 0.5

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



# 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?



- Quiz 3: For which  $\gamma$  are West and East equally good when in state d?

$$10\gamma = \gamma^3$$

# Infinite Utilities?!

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

- Solutions:

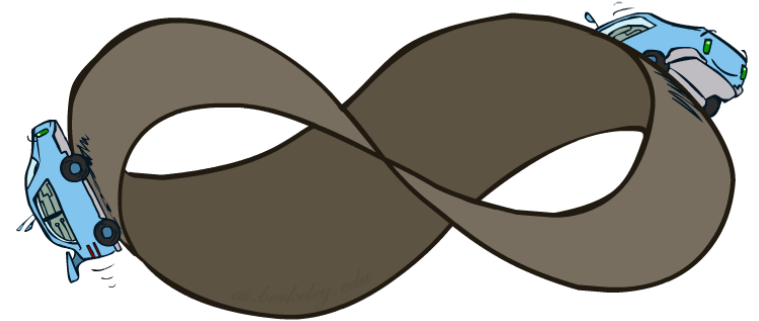
- Finite horizon: (similar to depth-limited search)

- Terminate episodes after a fixed T steps (e.g. life)
- Policy  $\pi$  depends on time left

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

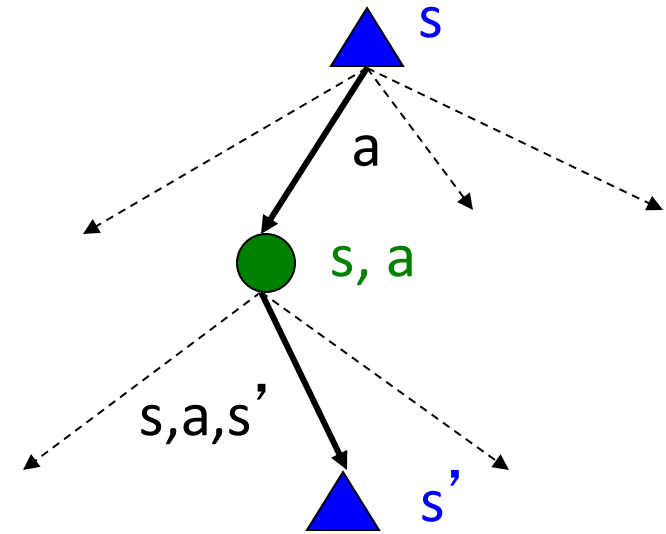
$$U([r_0, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max} / (1 - \gamma)$$

- 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)

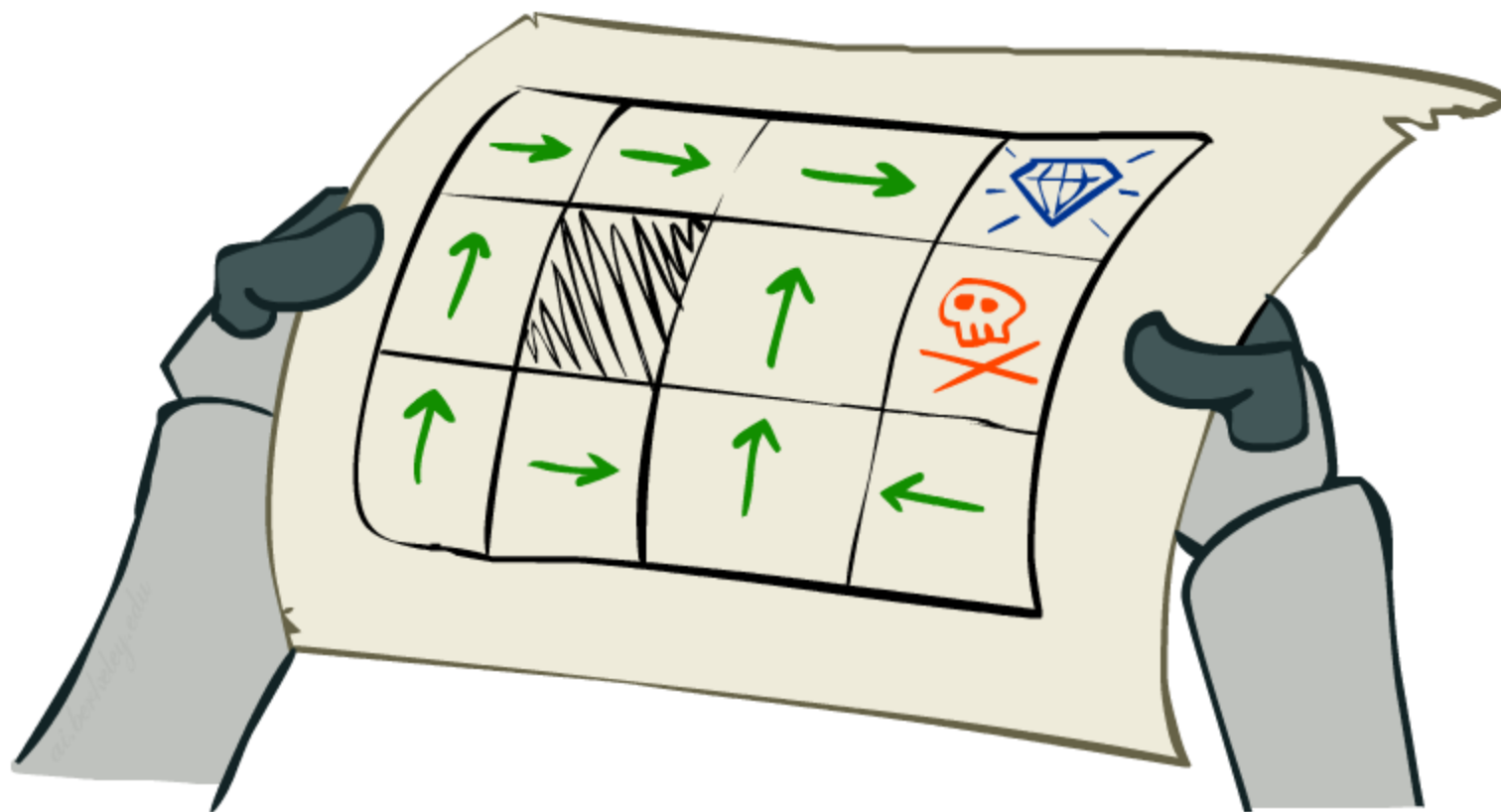


# Recap: Defining MDPs

- 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

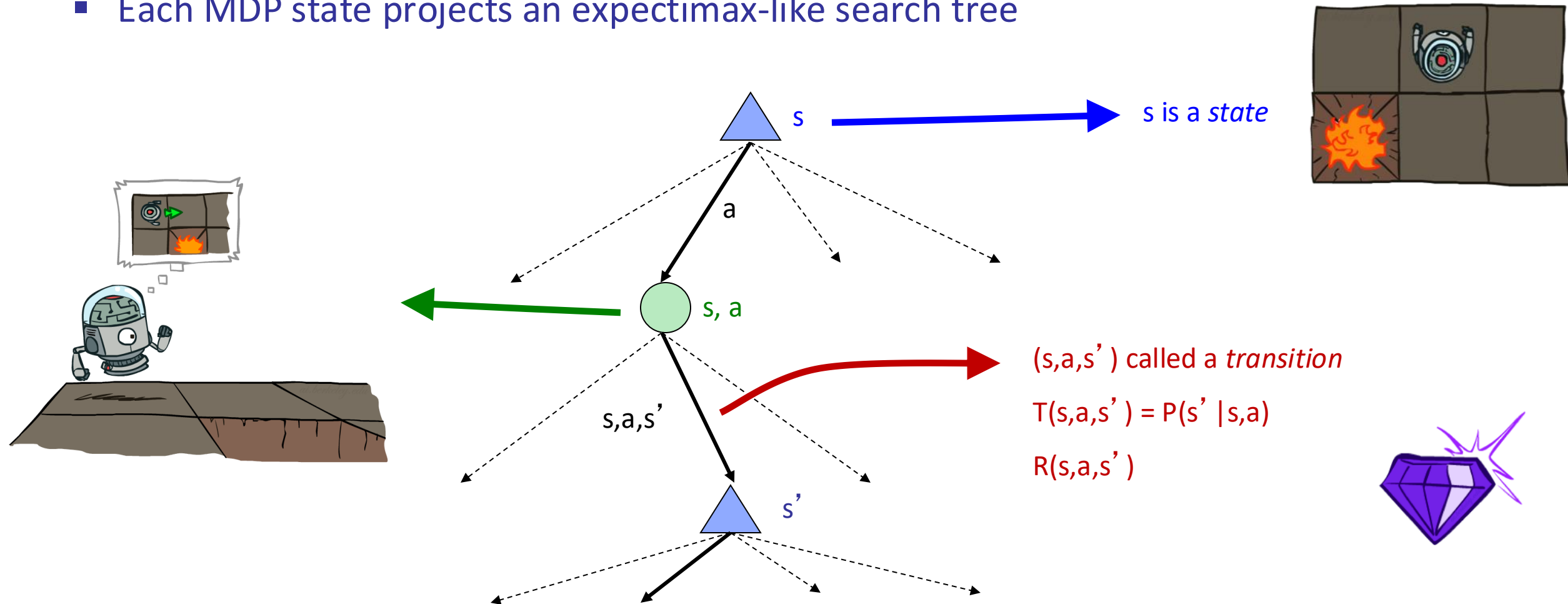


# Solving MDPs



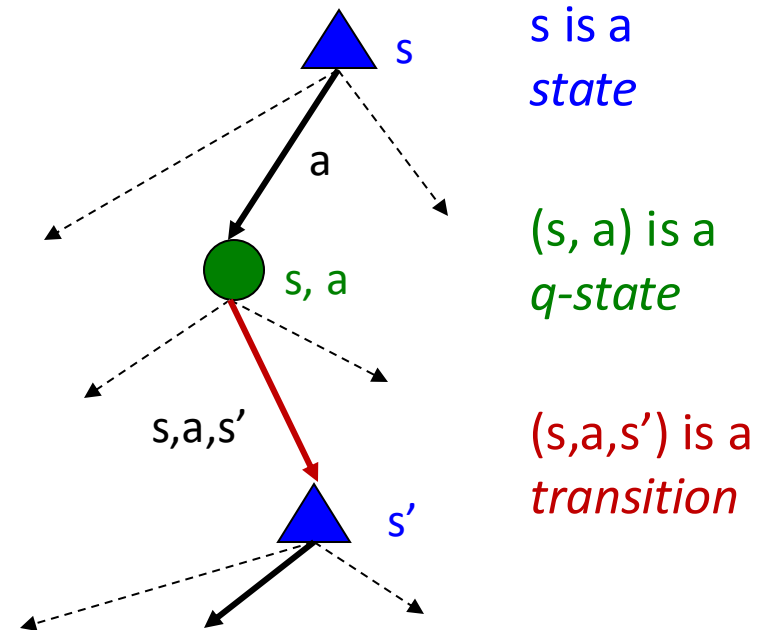
# MDP Search Trees

- Each MDP state projects an expectimax-like search tree



# 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  $s$



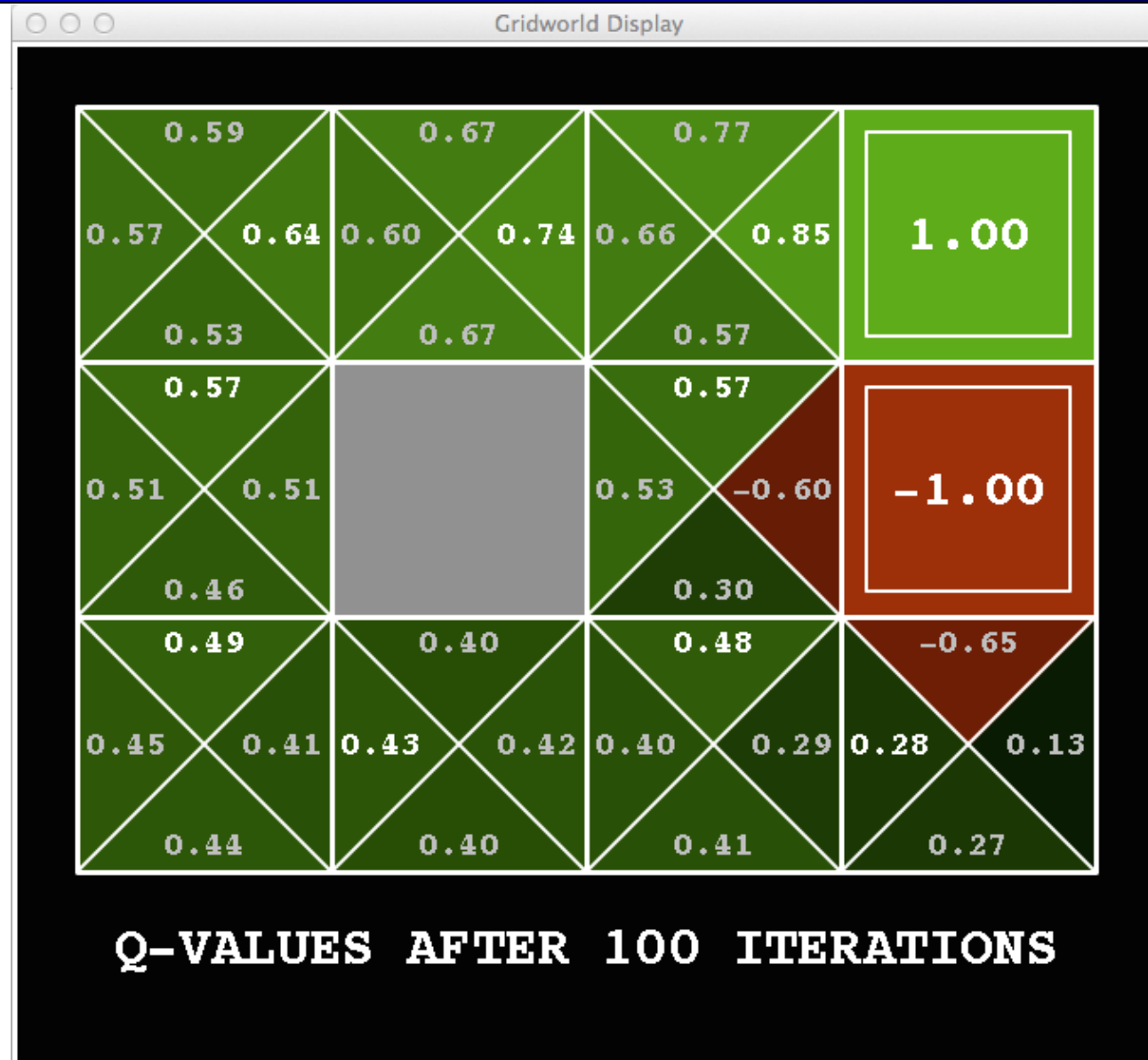


# Snapshot Gridworld V Values



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Snapshot of Gridworld Q Values



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Values of States (Bellman Equations)

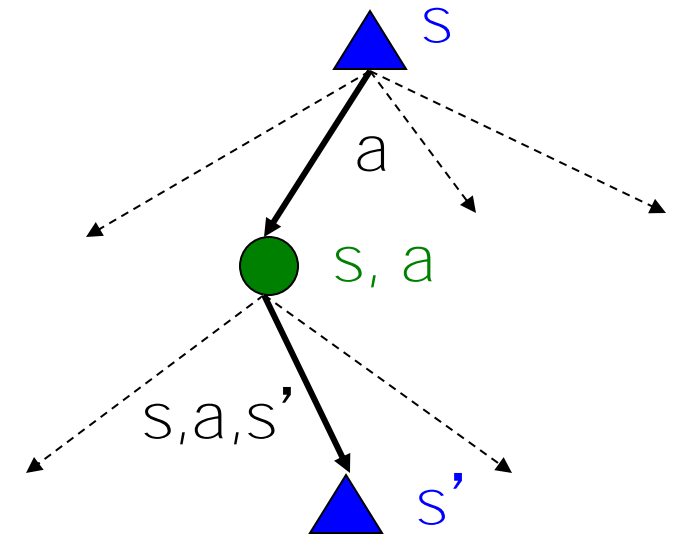
- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!

- Recursive definition of value:

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

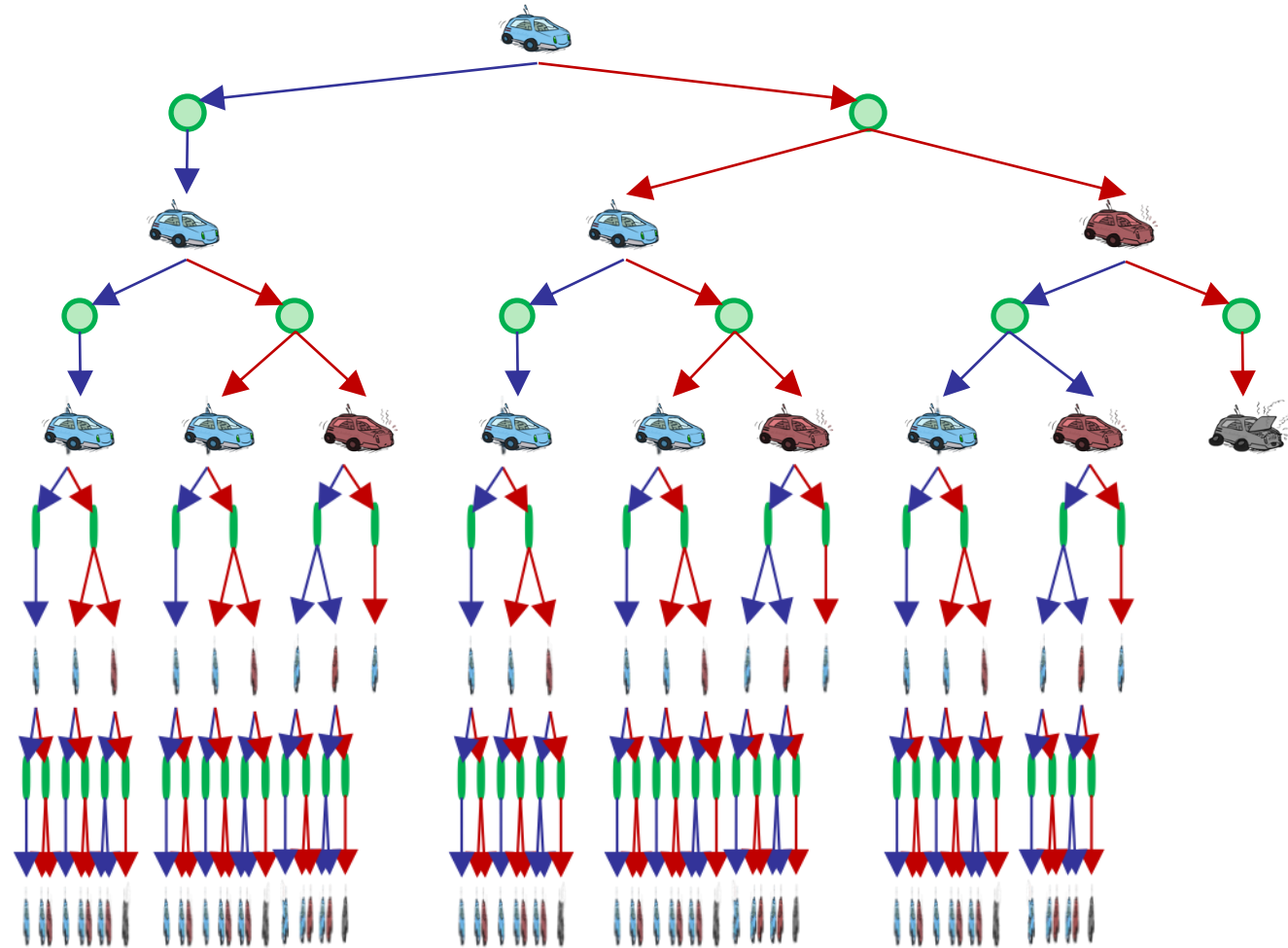
$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

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



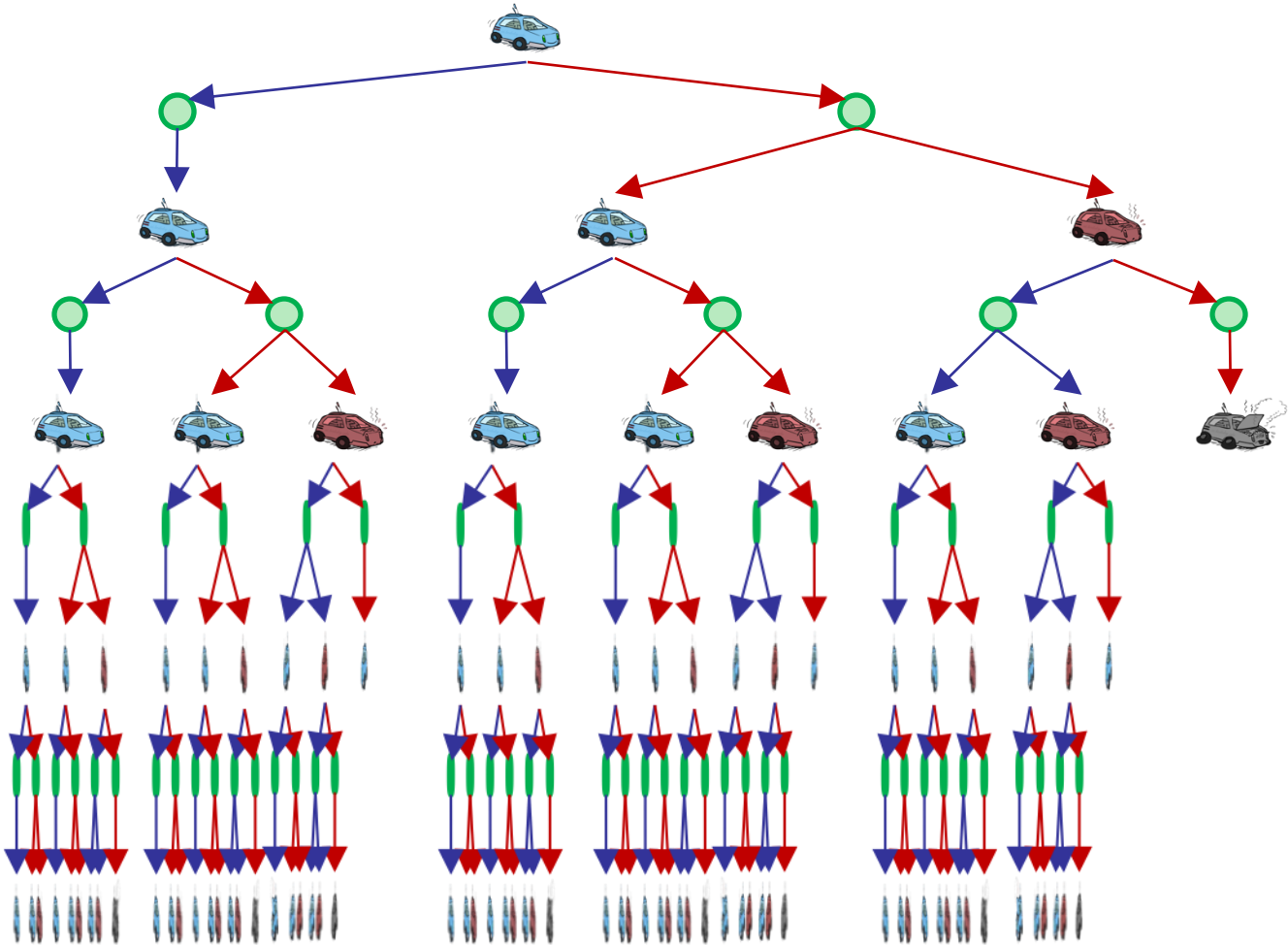


# Racing Search Tree



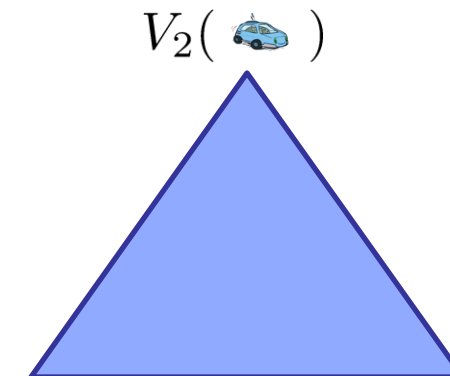
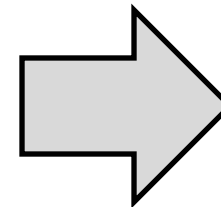
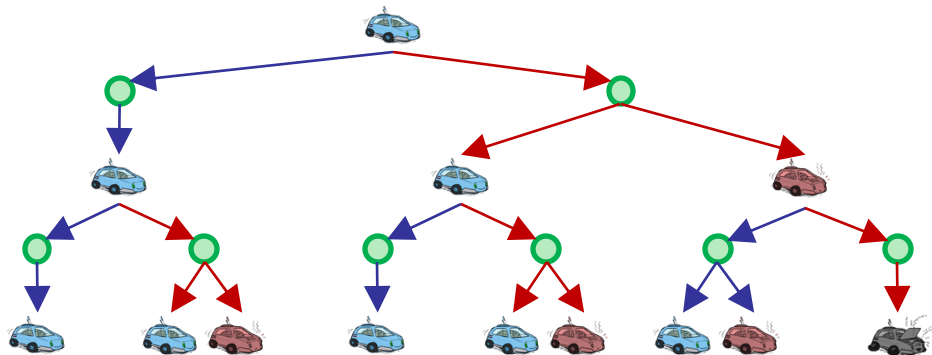
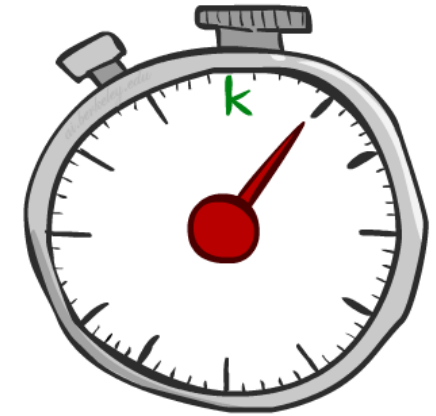
# Racing Search Tree

- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea quantities: Only compute needed 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  $\gamma < 1$

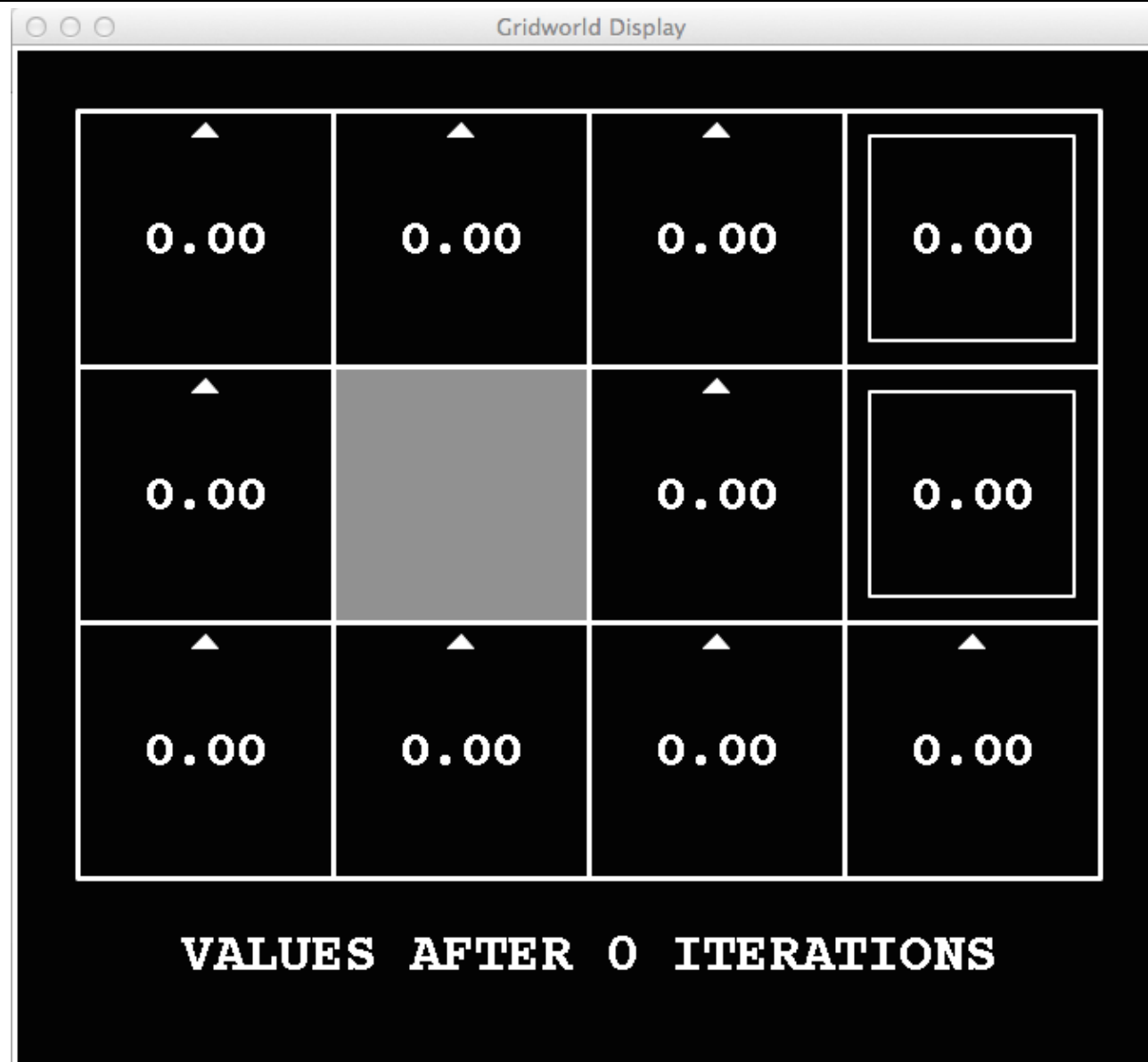


# Time-Limited Values

- 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



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# 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



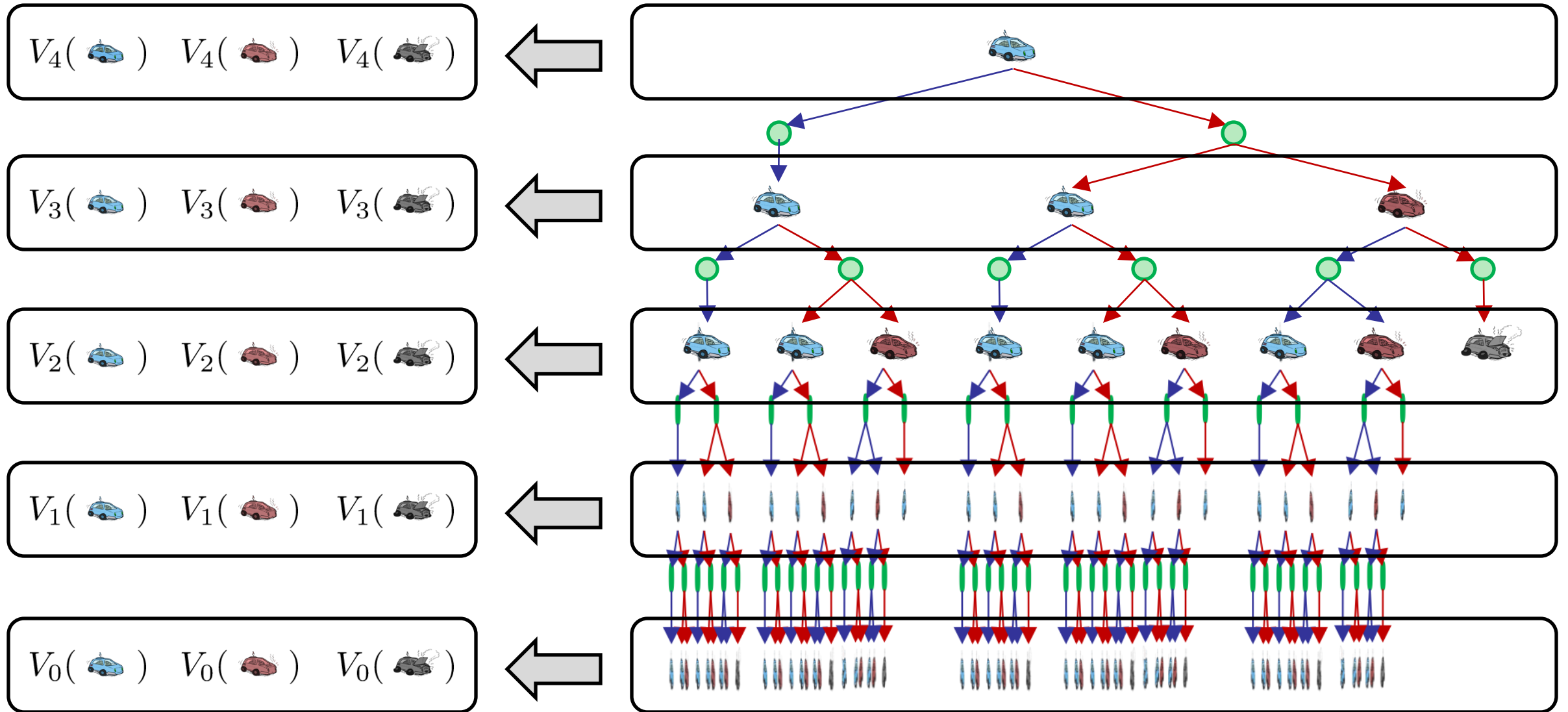
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=100



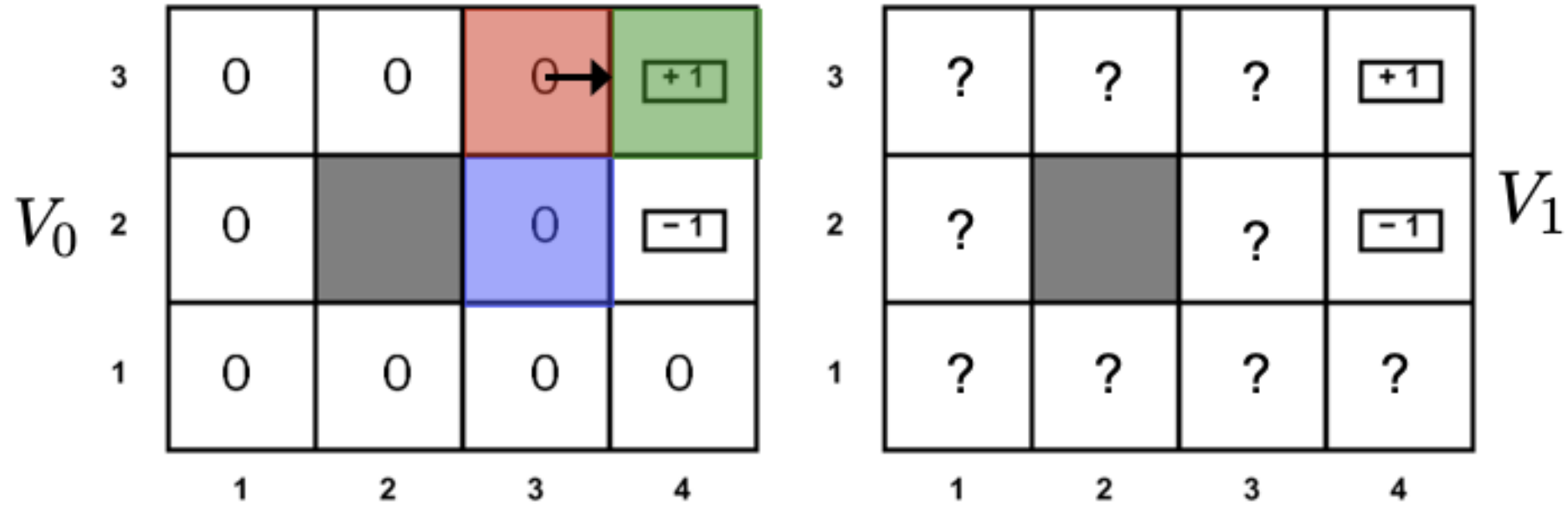
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Computing Time-Limited Values



# Bellman Updates

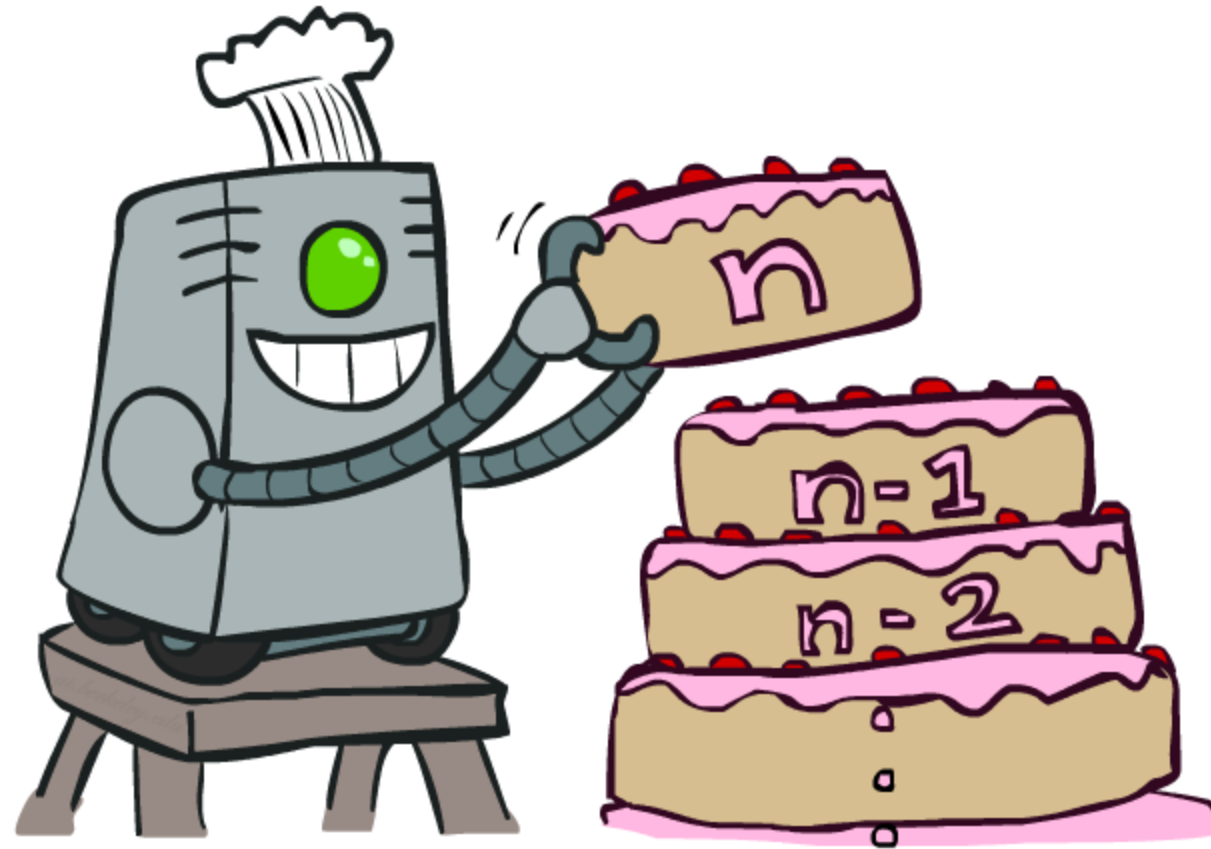
Example:  $\gamma=0.9$ , living  
reward=0, noise=0.2



$$V_{i+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_i(s')] = \max_a Q_{i+1}(s, a)$$

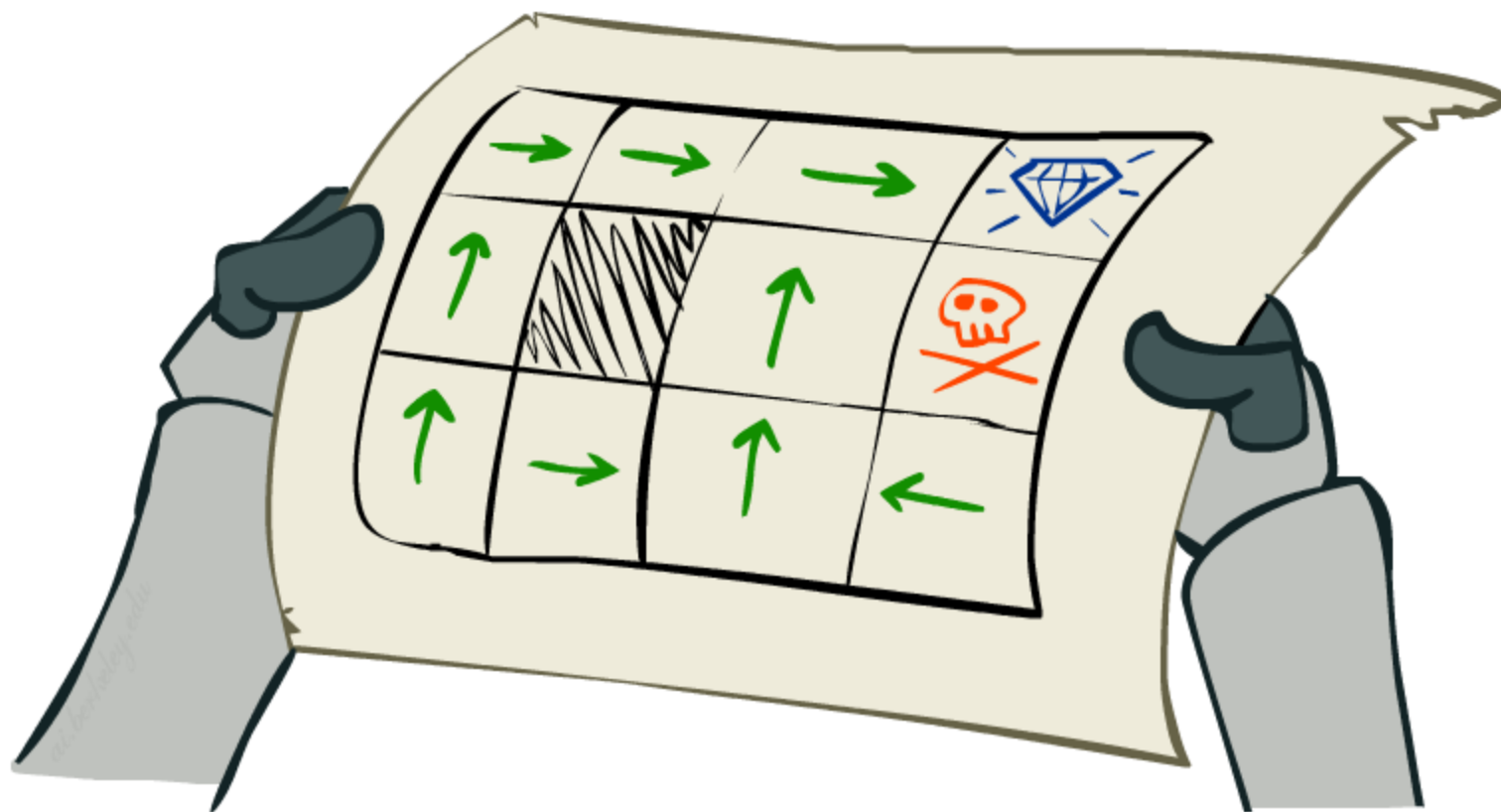
$$Q_1(\langle 3, 3 \rangle, \text{right}) = \sum_{s'} T(\langle 3, 3 \rangle, \text{right}, s') [R(\langle 3, 3 \rangle, \text{right}, s') + \gamma V_i(s')] \\ = 0.8 * [0.0 + 0.9 * 1.0] + 0.1 * [0.0 + 0.9 * 0.0] + 0.1 * [0.0 + 0.9 * 0.0]$$

# Value Iteration





# Solving MDPs

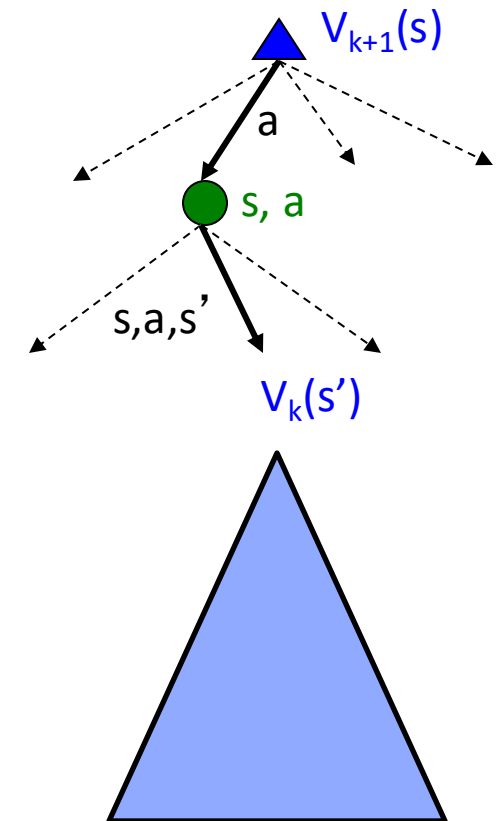


# Value Iteration

- 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



$V_2$

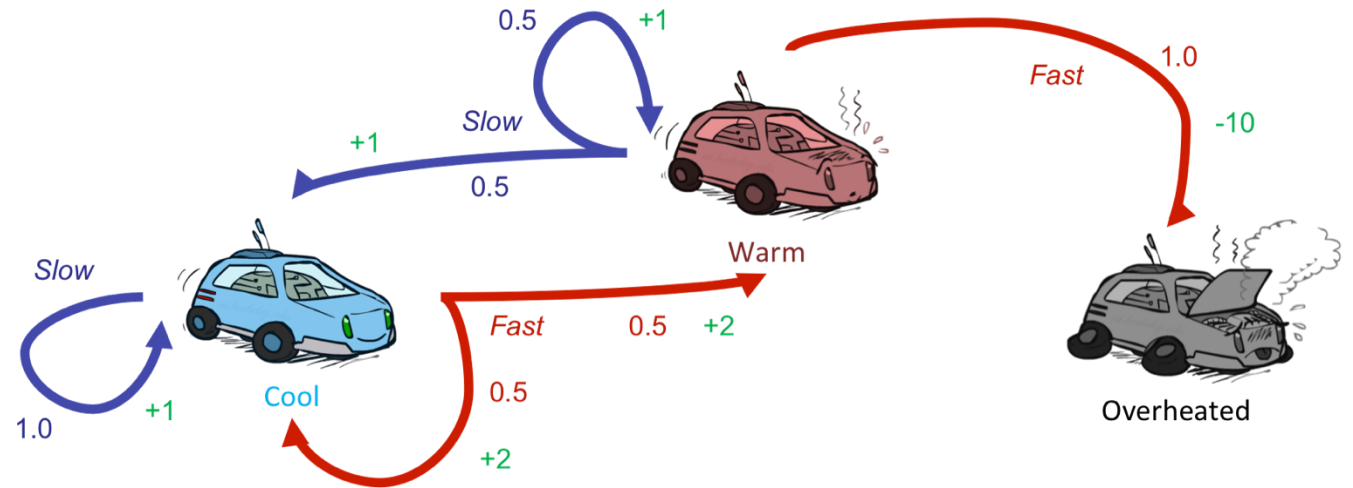


$V_1$

S: 1  
F:  $.5*2+.5*2=2$

$V_0$

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



$V_2$



$V_1$

2

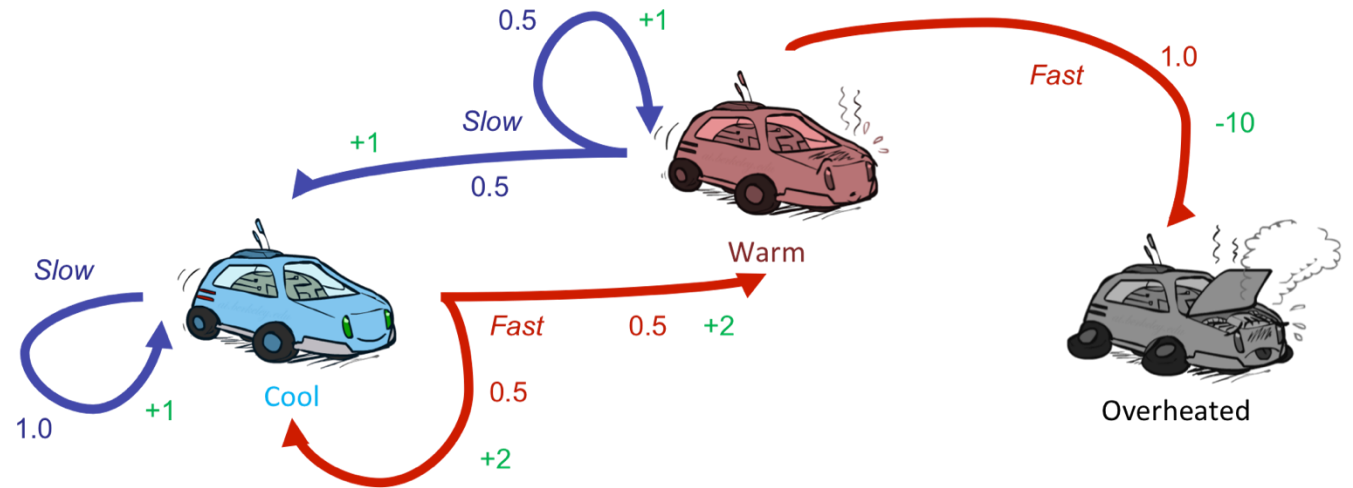
S:  $.5*1+.5*1=1$   
F: -10

$V_0$

0

0

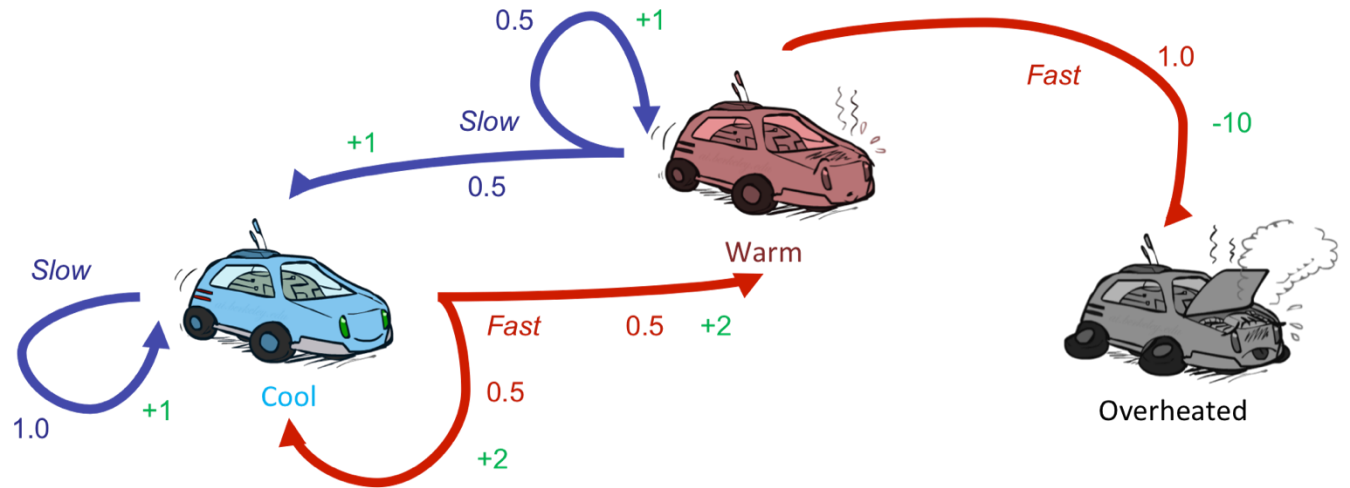
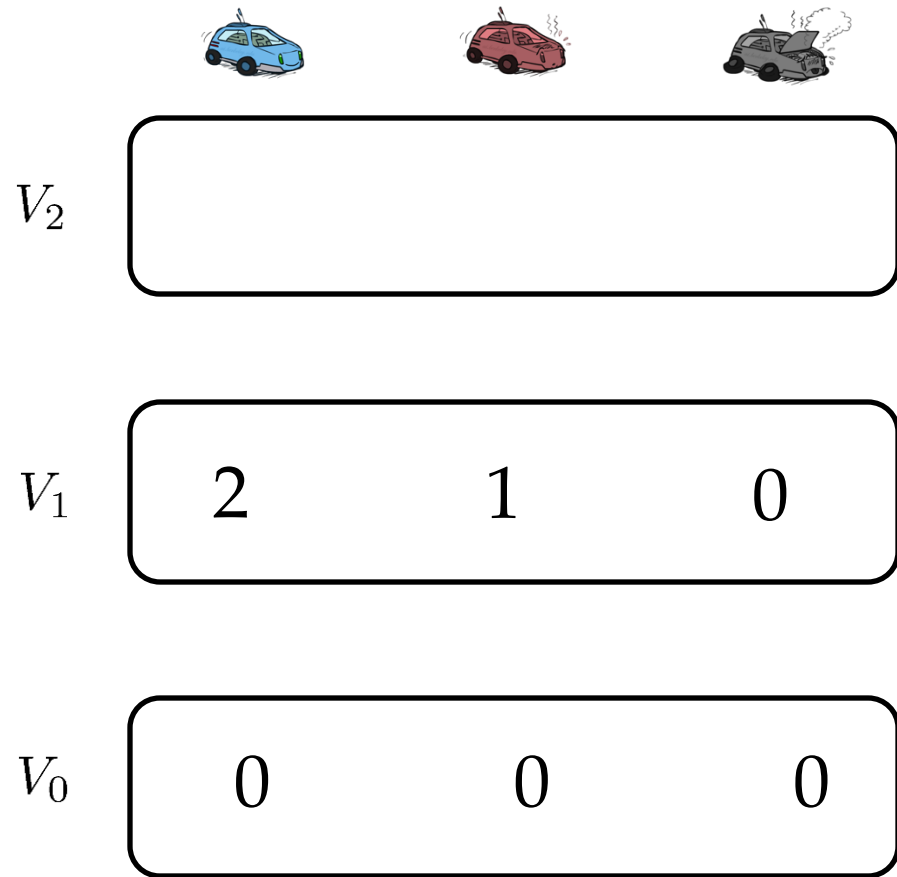
0



*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



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



$V_2$

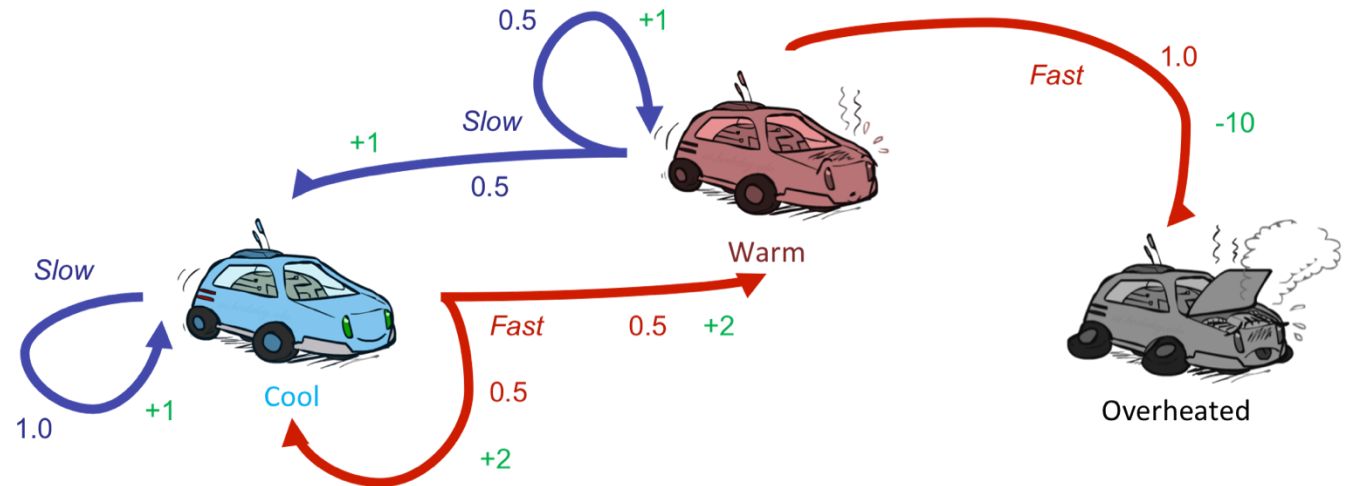
S:  $1+2=3$   
 F:  $.5*(2+2)+.5*(2+1)=3.5$

$V_1$

2      1      0

$V_0$




0      0      0

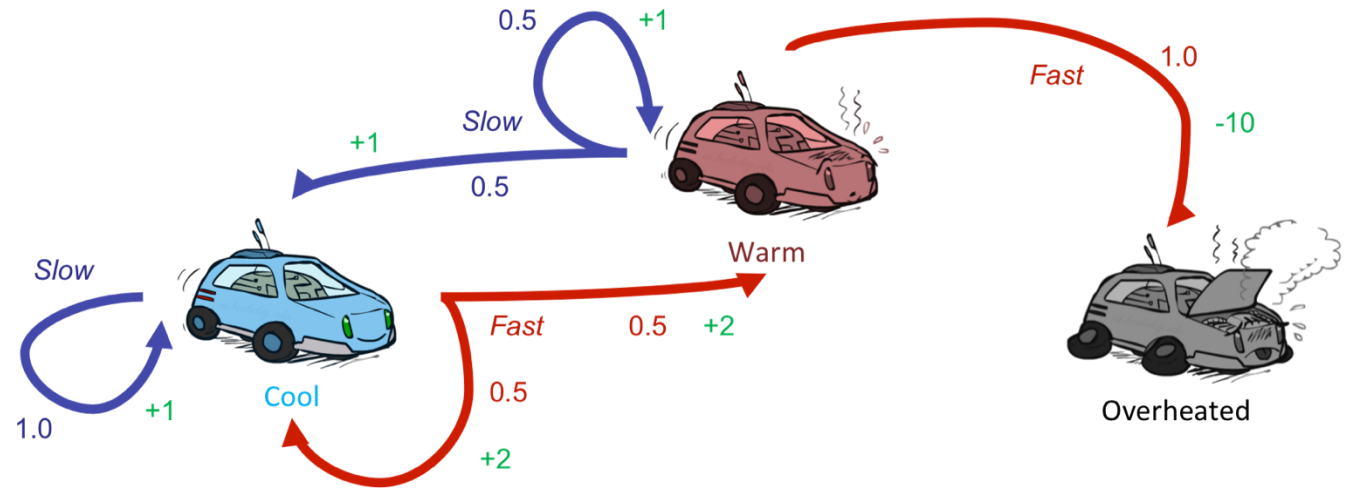


*Assume no discount!*

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

# Example: Value Iteration

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



Assume no discount!

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

# Value Iteration

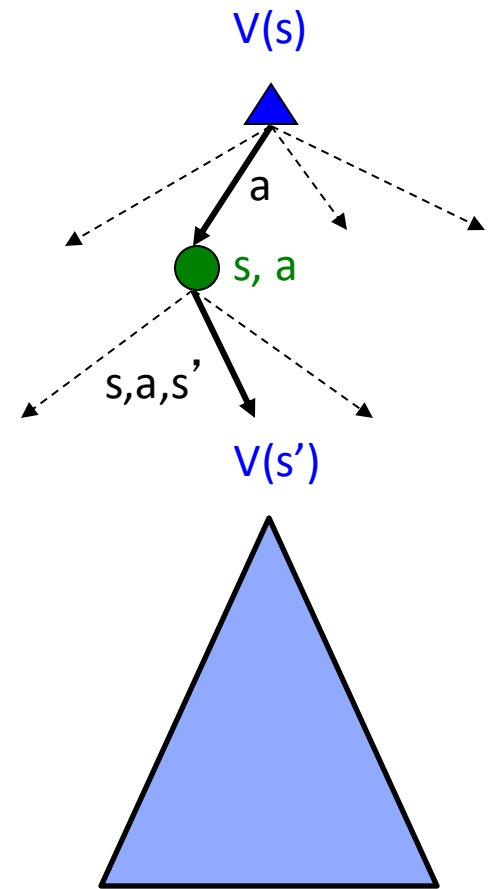
- Bellman equations **characterize** the optimal values:

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

- Value iteration **computes** them:

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

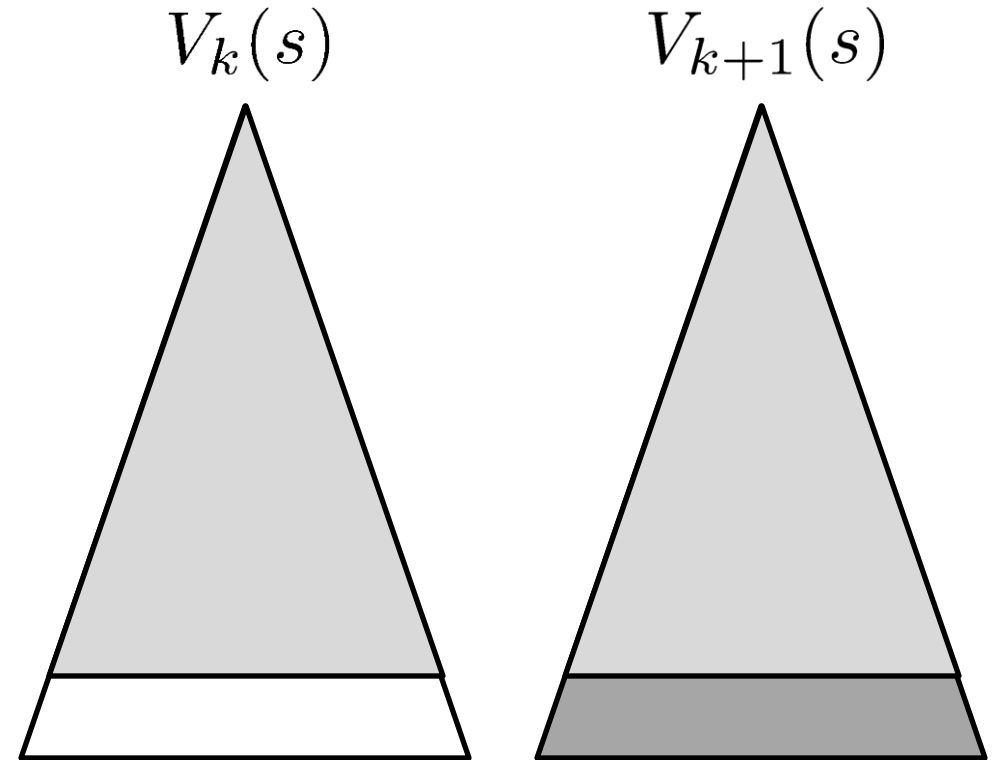
- Value iteration is just a fixed point solution method
  - ... though the  $V_k$  vectors are also interpretable as time-limited values



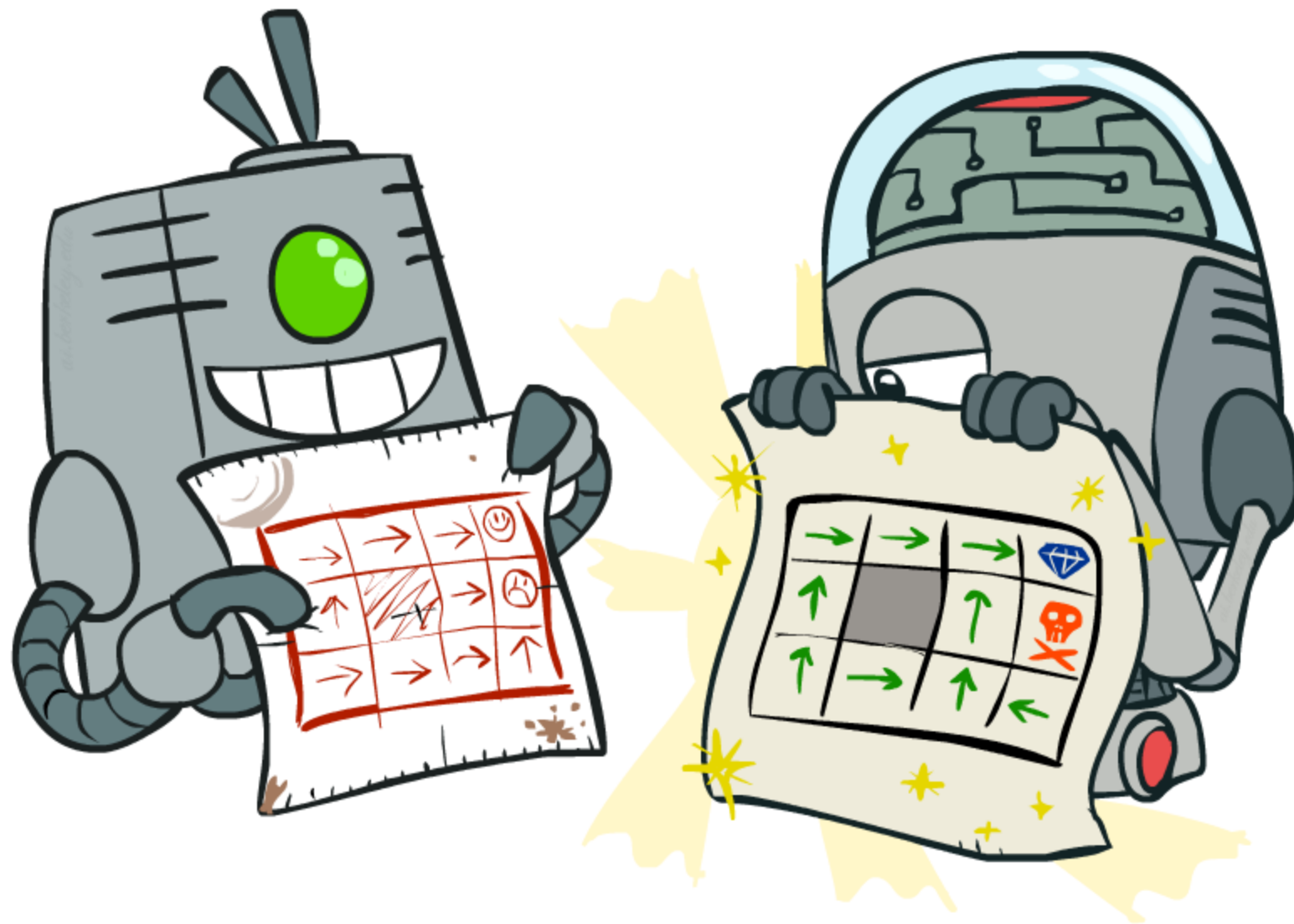


# Convergence\*

- 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  $\gamma^k$  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



# Policy Methods

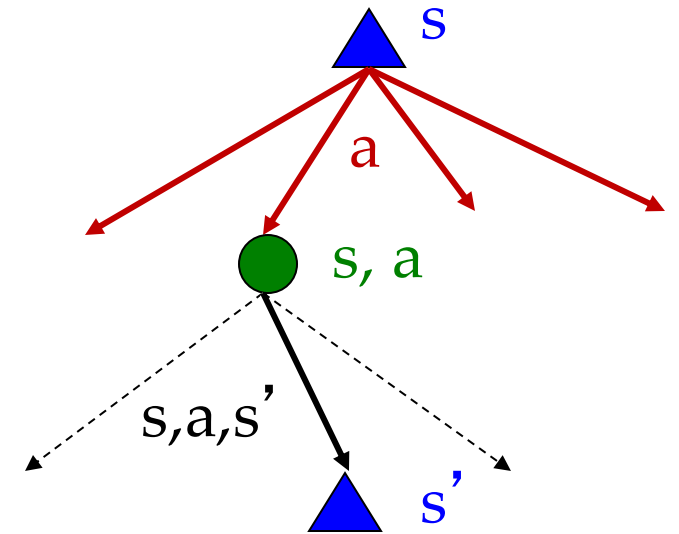


# Problems with Value Iteration

- 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
- Problem 3: The policy often converges long before the values



# k=12



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=100



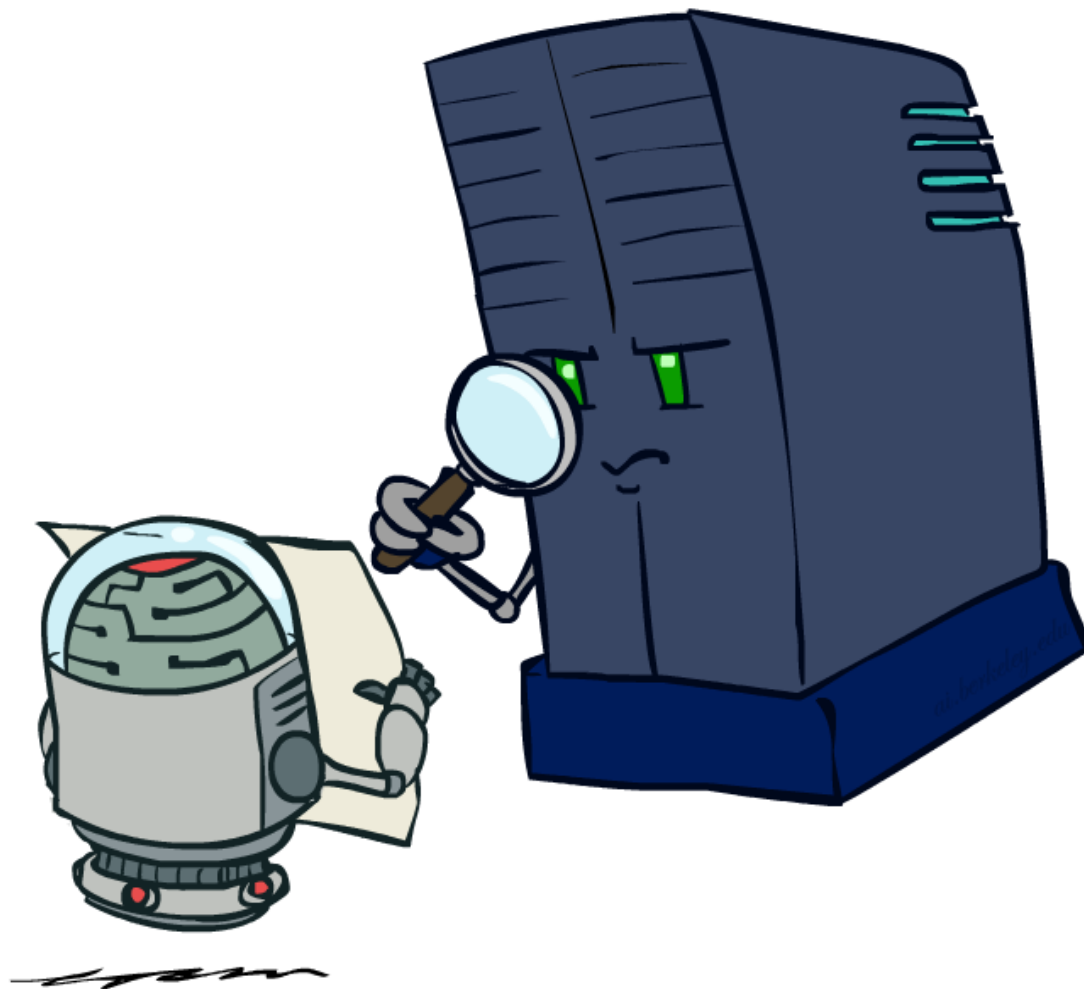
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# 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

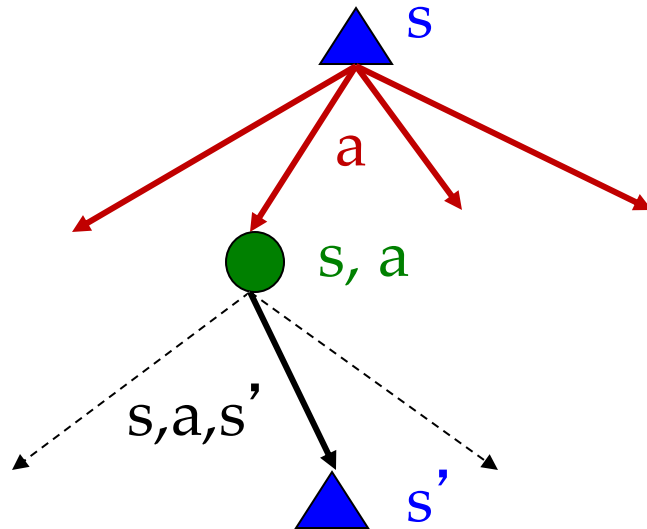
# Policy Evaluation

---

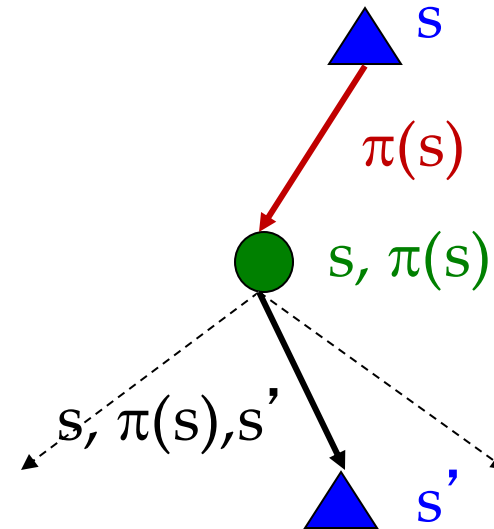


# Fixed Policies

Do the optimal action



Do what  $\pi$  says to do



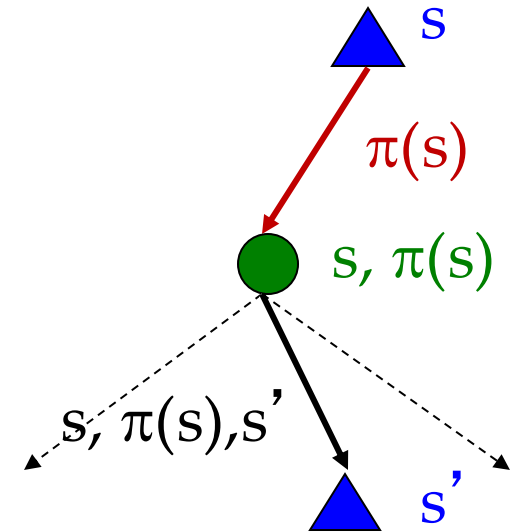
- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy  $\pi(s)$ , then the tree would be simpler – only one action per state
  - ... though the tree's value would depend on which policy we fixed



# Utilities for a Fixed Policy

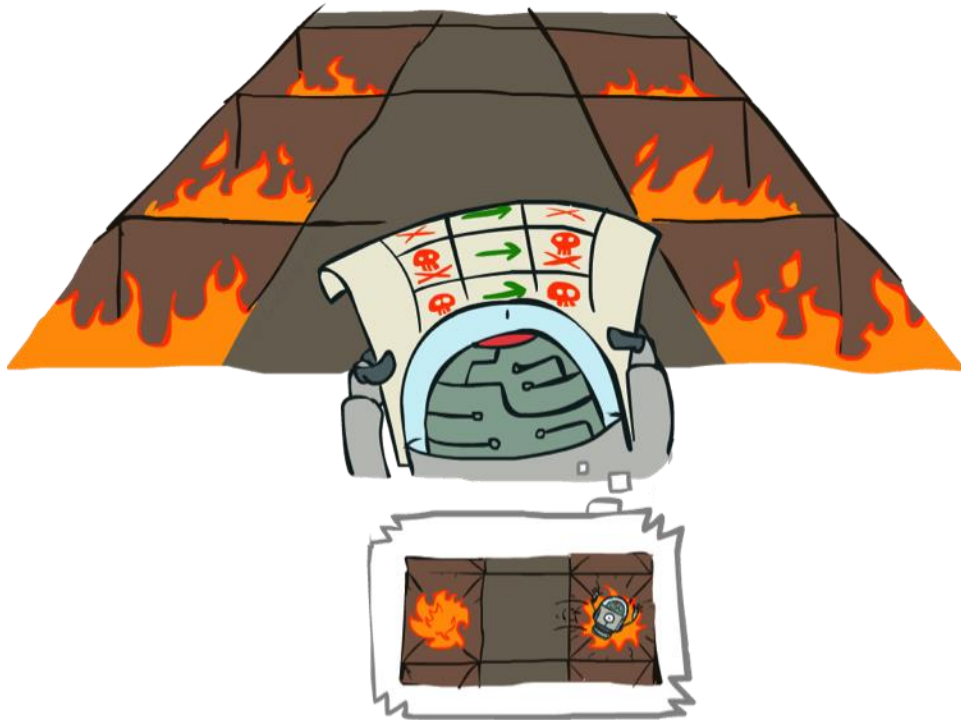
- Another basic operation: compute the utility of a state  $s$  under a fixed (generally non-optimal) policy
- Define the utility of a state  $s$ , under a fixed policy  $\pi$ :  
 $V^\pi(s)$  = expected total discounted rewards starting in  $s$  and following  $\pi$
- Recursive relation (one-step look-ahead / Bellman equation):

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

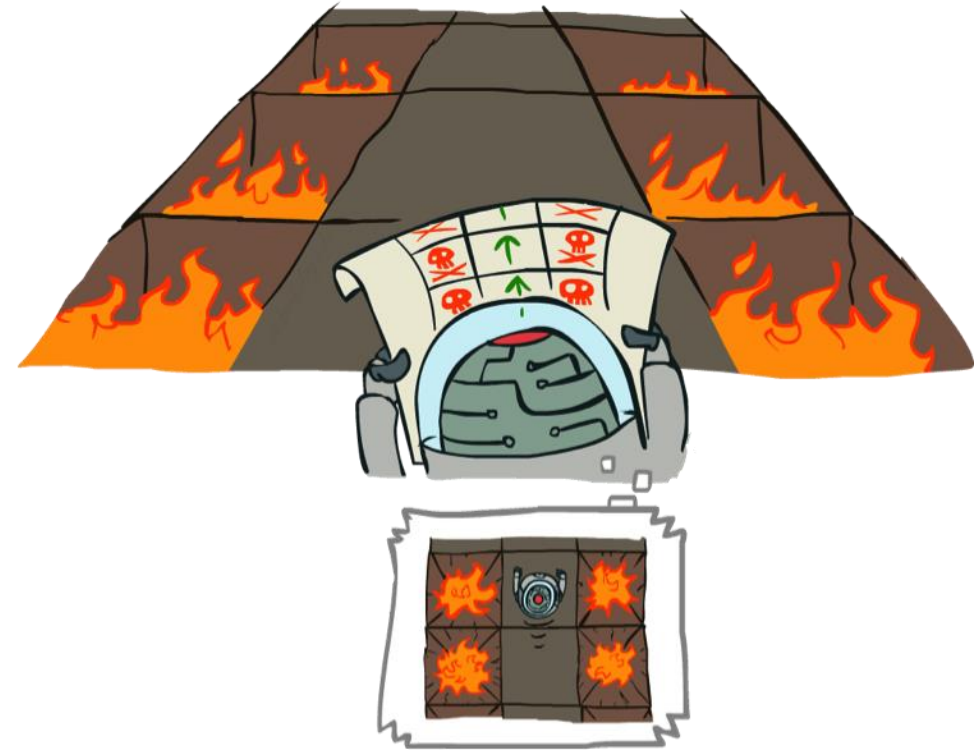


# Example: Policy Evaluation

Always Go Right



Always Go Forward



# Example: Policy Evaluation

Always Go Right



Always Go Forward

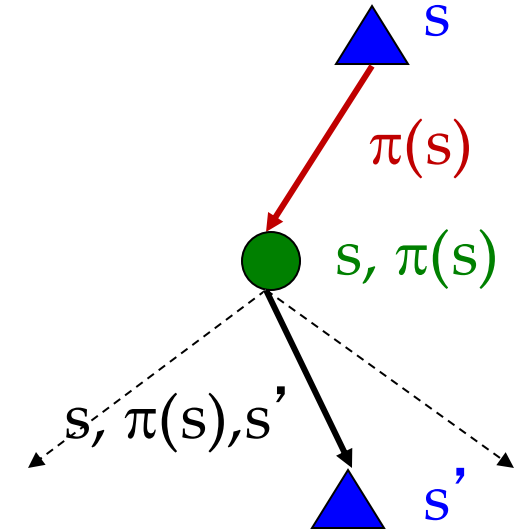


# Policy Evaluation

- How do we calculate the  $V$ 's for a fixed policy  $\pi$ ?
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

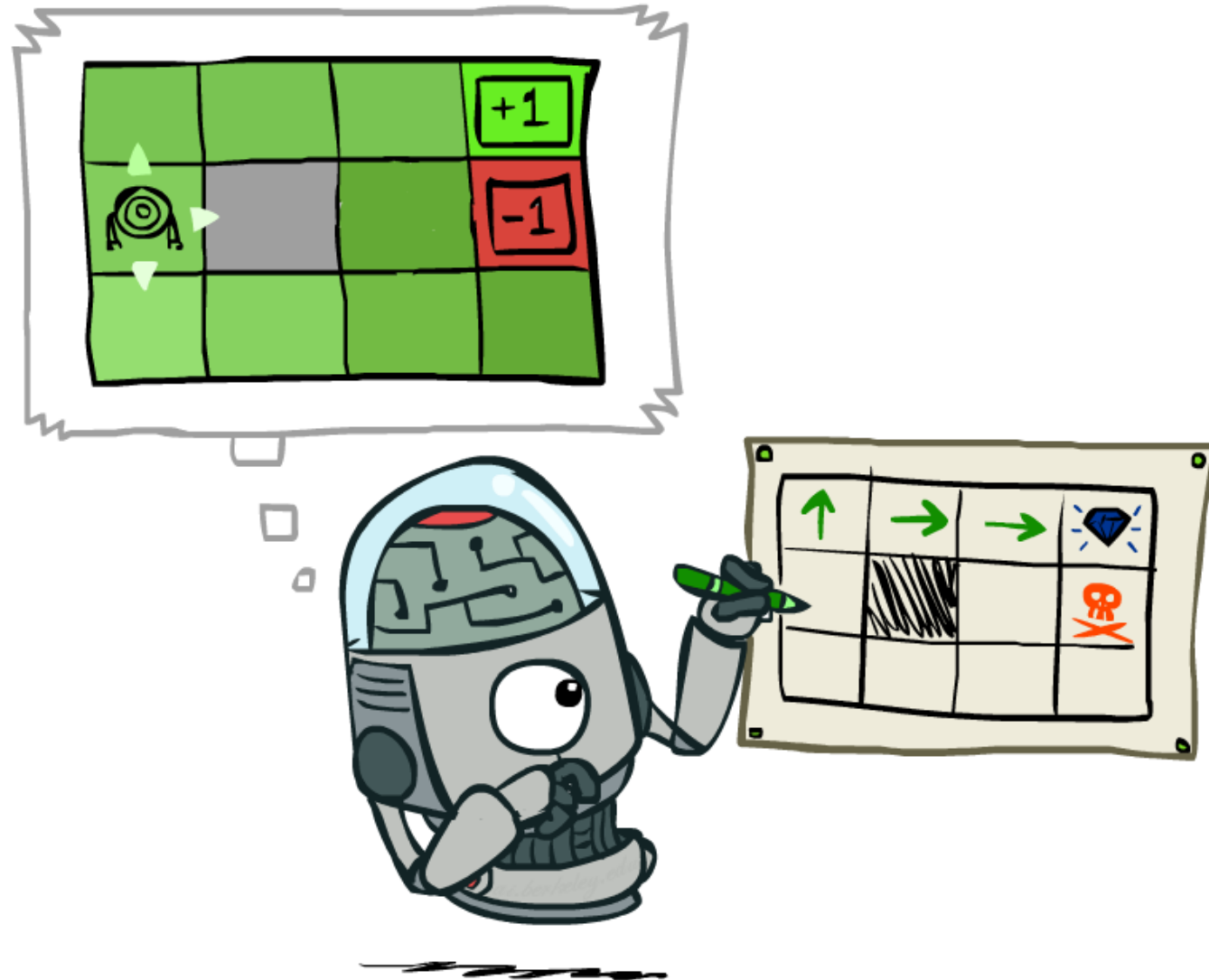
$$V_0^\pi(s) = 0$$

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



- 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)

# Policy Extraction



# 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)



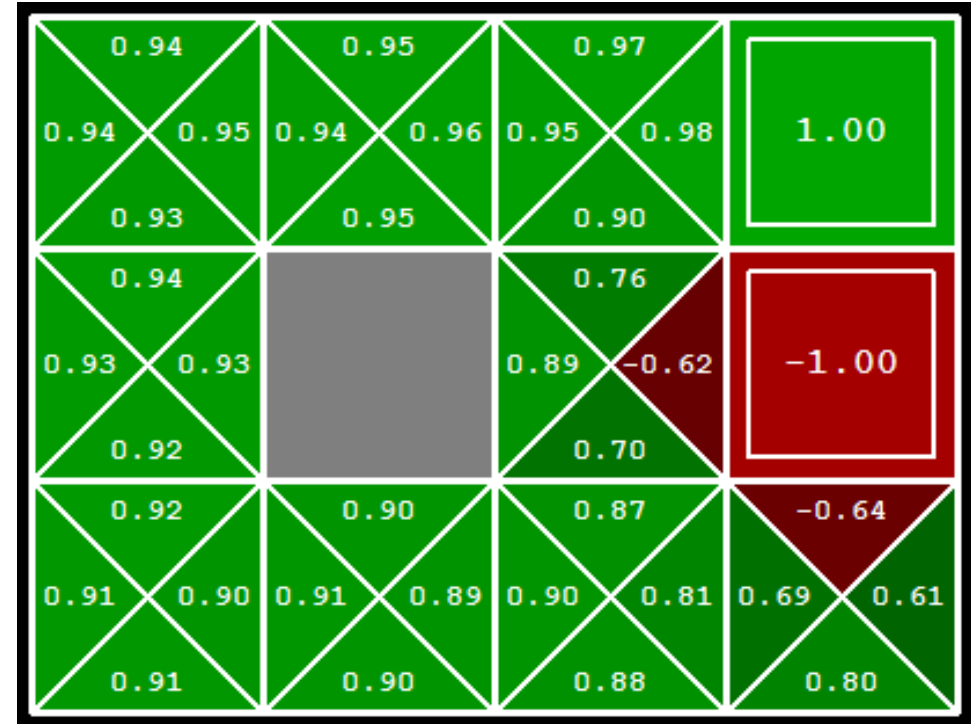
$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

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

# Computing Actions from Q-Values

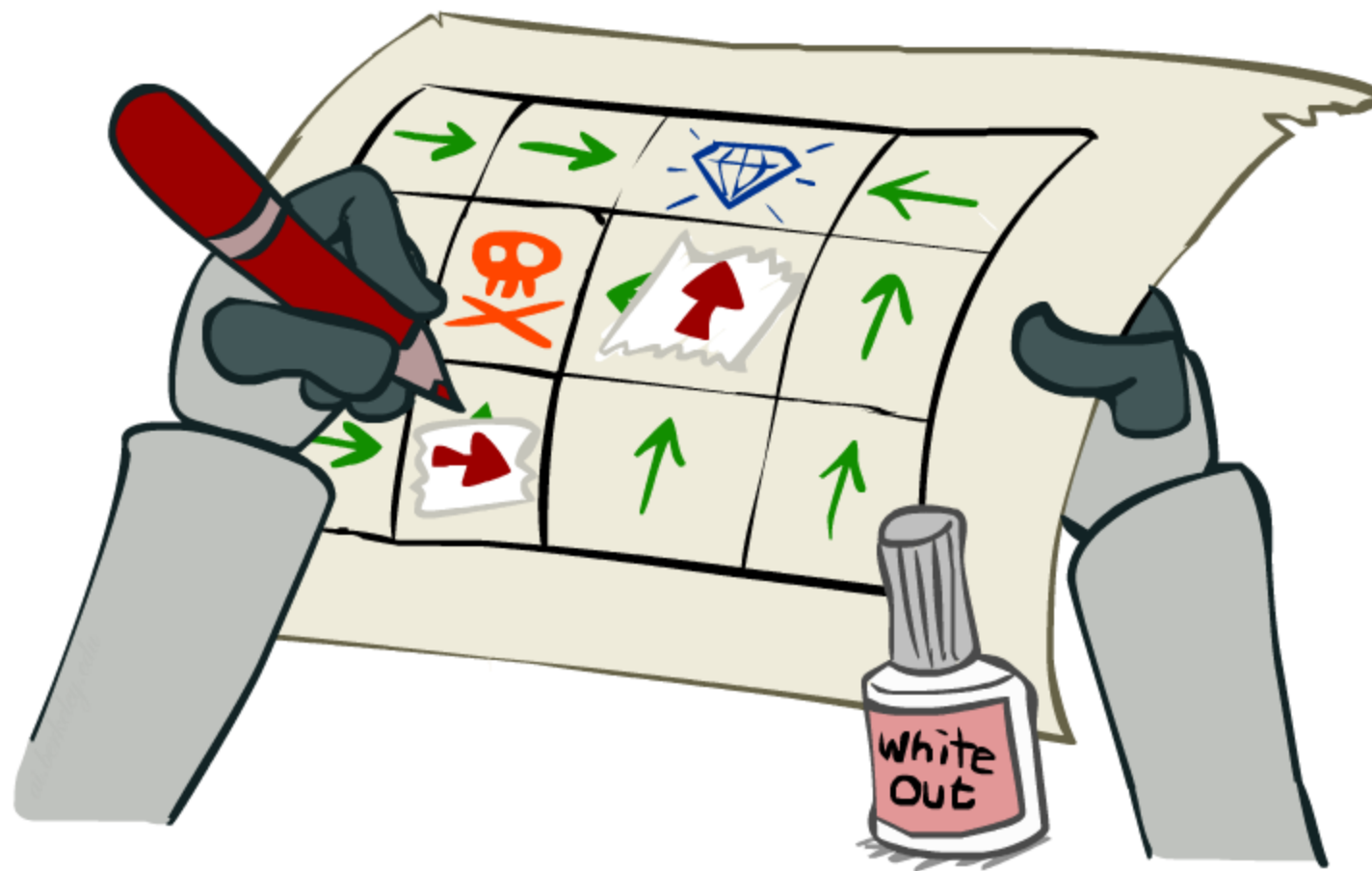
- Let's imagine we have the optimal q-values:
- How should we act?
  - Completely trivial to decide!

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



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

# Policy Iteration





# 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  $\pi$ , find values with policy evaluation:
  - Iterate until values converge:

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

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

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

# Comparison

---

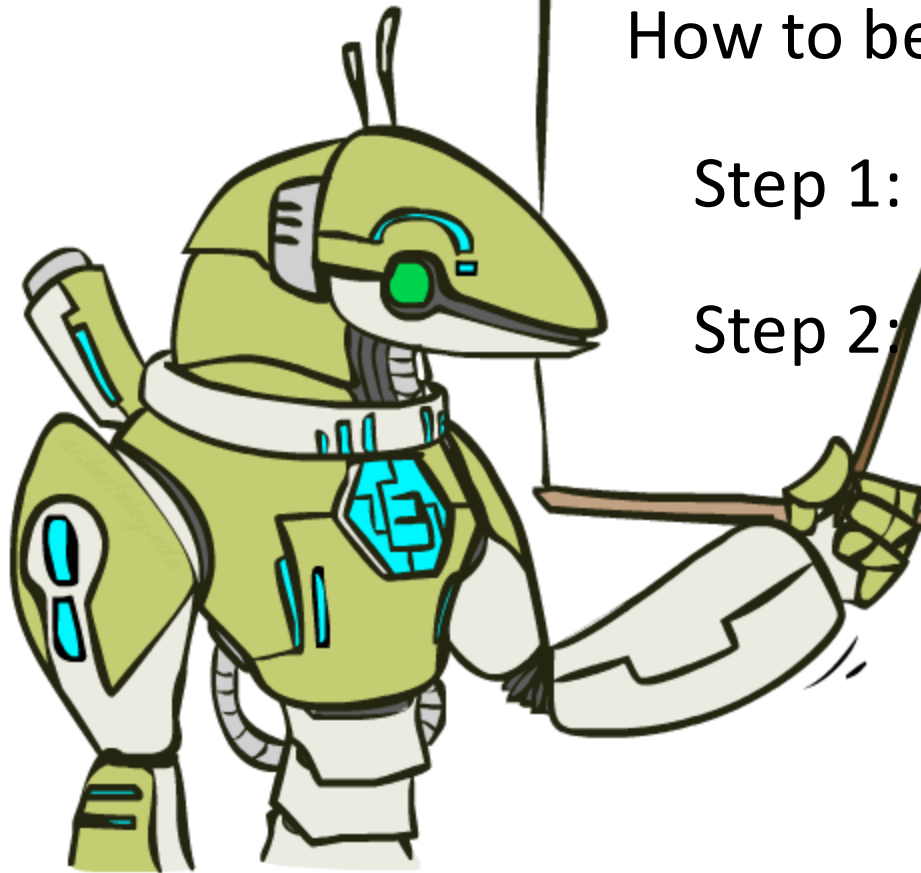
- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do 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

# The Bellman Equations



How to be optimal:

Step 1: Take correct first action

Step 2: Keep being optimal