Lecture 6: Markov Decision Processes

Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

https://shuaili8.github.io

https://shuaili8.github.io/Teaching/CS410/index.html

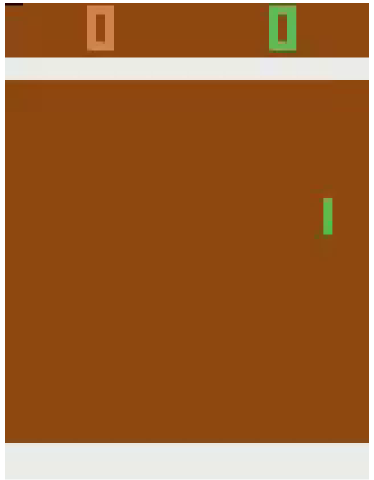
Part of slide credits: CMU AI & http://ai.berkeley.edu

Recent Progress by Deep Reinforcement Learning

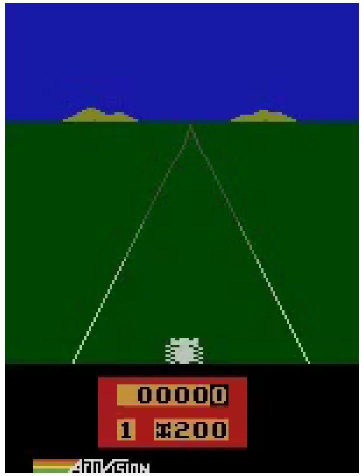
Deep Reinforcement Learning

2013

Atari (DQN)
[Deepmind]

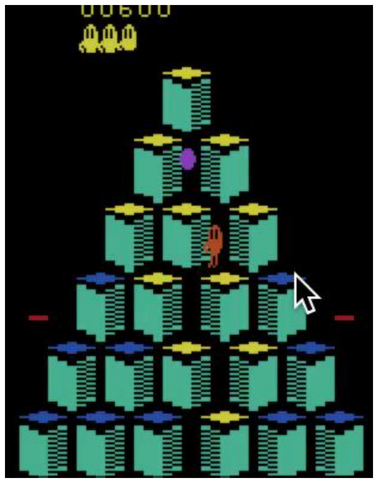


Pong



Enduro

Beamrider

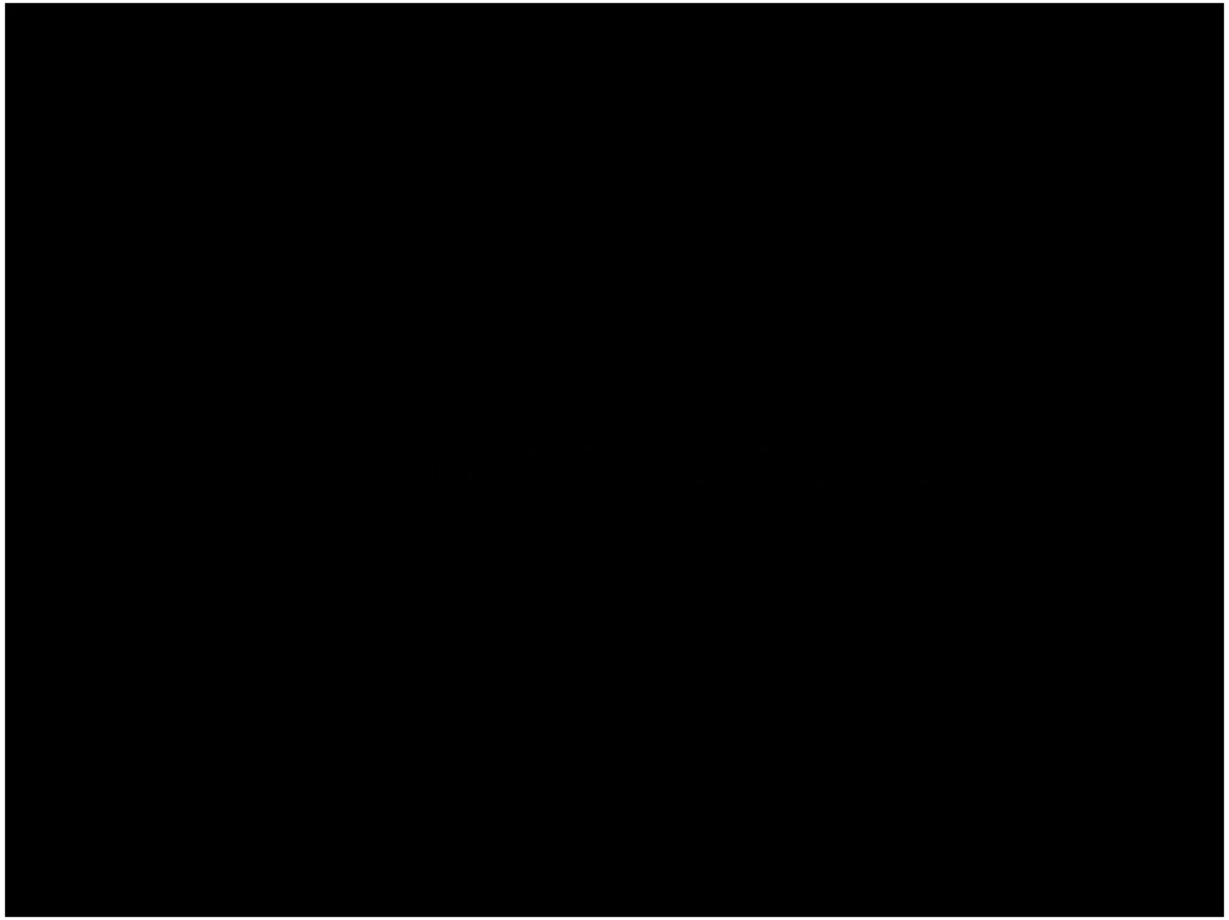


Q*bert

Deep Reinforcement Learning 2

2013 **Atari (DQN)**
[Deepmind]

2015 **Human-level control**
[Deepmind]



Trained separate DQN agents for 50 different Atari games, without any prior knowledge of the game rules

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529.

Deep Reinforcement Learning 3

2013 **Atari (DQN)**
[Deepmind]

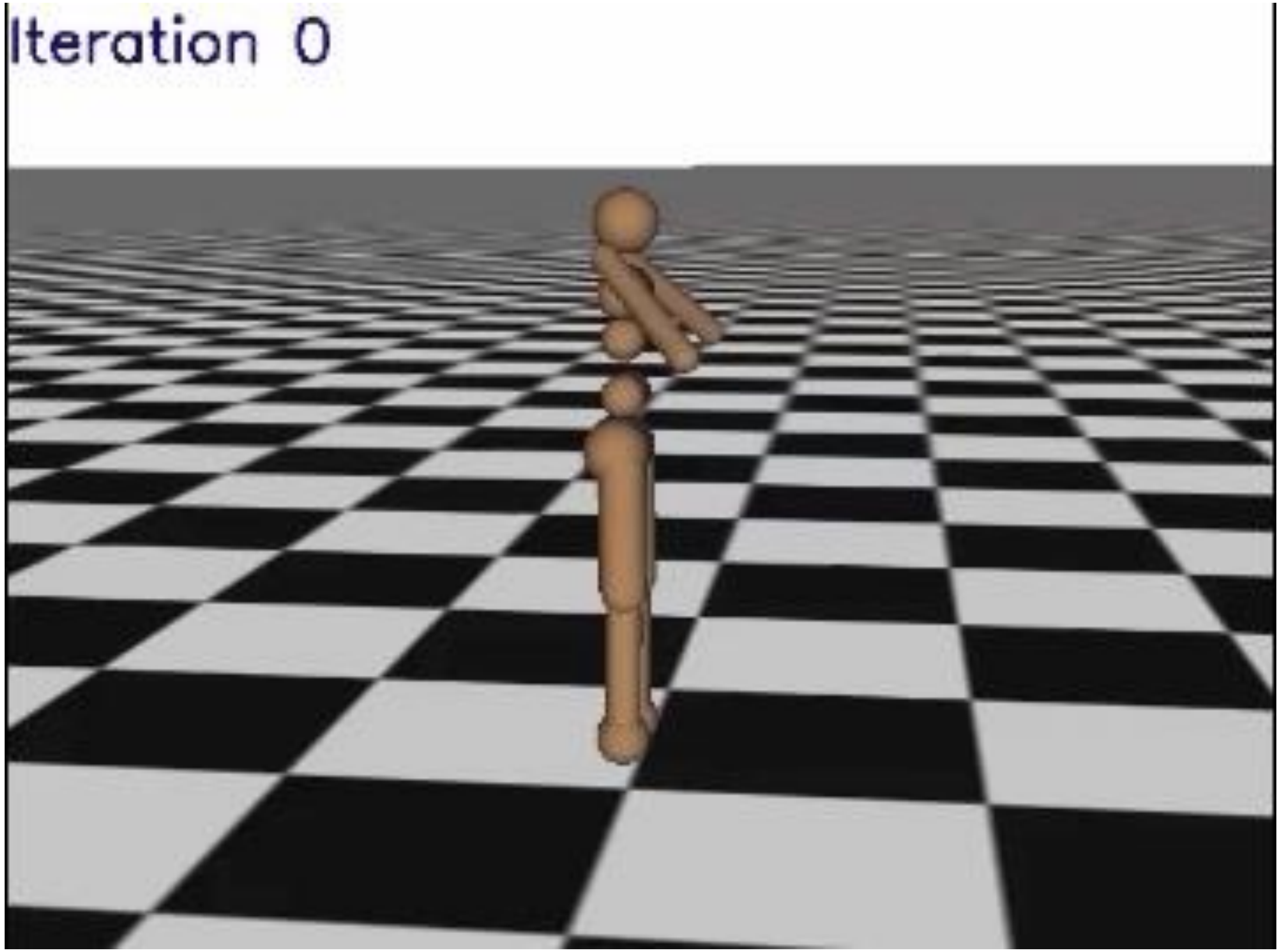
2015 **Human-level control**
[Deepmind]

AlphaGo
[Deepmind]

AlphaGo Silver et al, Nature 2015
AlphaGoZero Silver et al, Nature 2017
AlphaZero Silver et al, 2017
Tian et al, 2016; Maddison et al, 2014; Clark et al, 2015

Deep Reinforcement Learning 4

2013	**Atari (DQN)** **[Deepmind]**
2015	**Human-level control** **[Deepmind]**
	AlphaGo **[Deepmind]**
2016	**3D locomotion (TRPO+GAE)** **[Berkeley]**



[Schulman, Moritz, Levine, Jordan, Abbeel, ICLR 2016]

Deep Reinforcement Learning 5

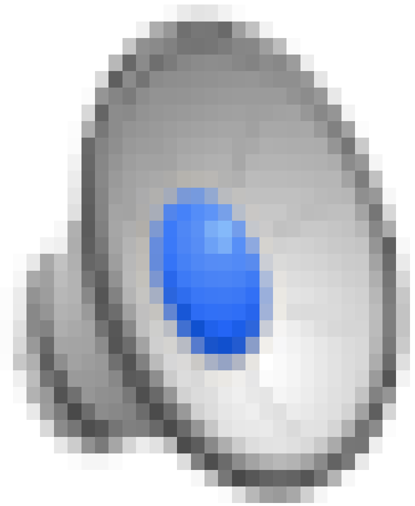
2013 **Atari (DQN)**
[Deepmind]

2015 **Human-level control**
[Deepmind]

AlphaGo
[Deepmind]

2016 **3D locomotion (TRPO+GAE)**
[Berkeley]

Real Robot Manipulation (GPS)
[Berkeley]



[Levine*, Finn*, Darrell, Abbeel, JMLR 2016]

Deep Reinforcement Learning 6

2013 **Atari (DQN)**
[Deepmind]

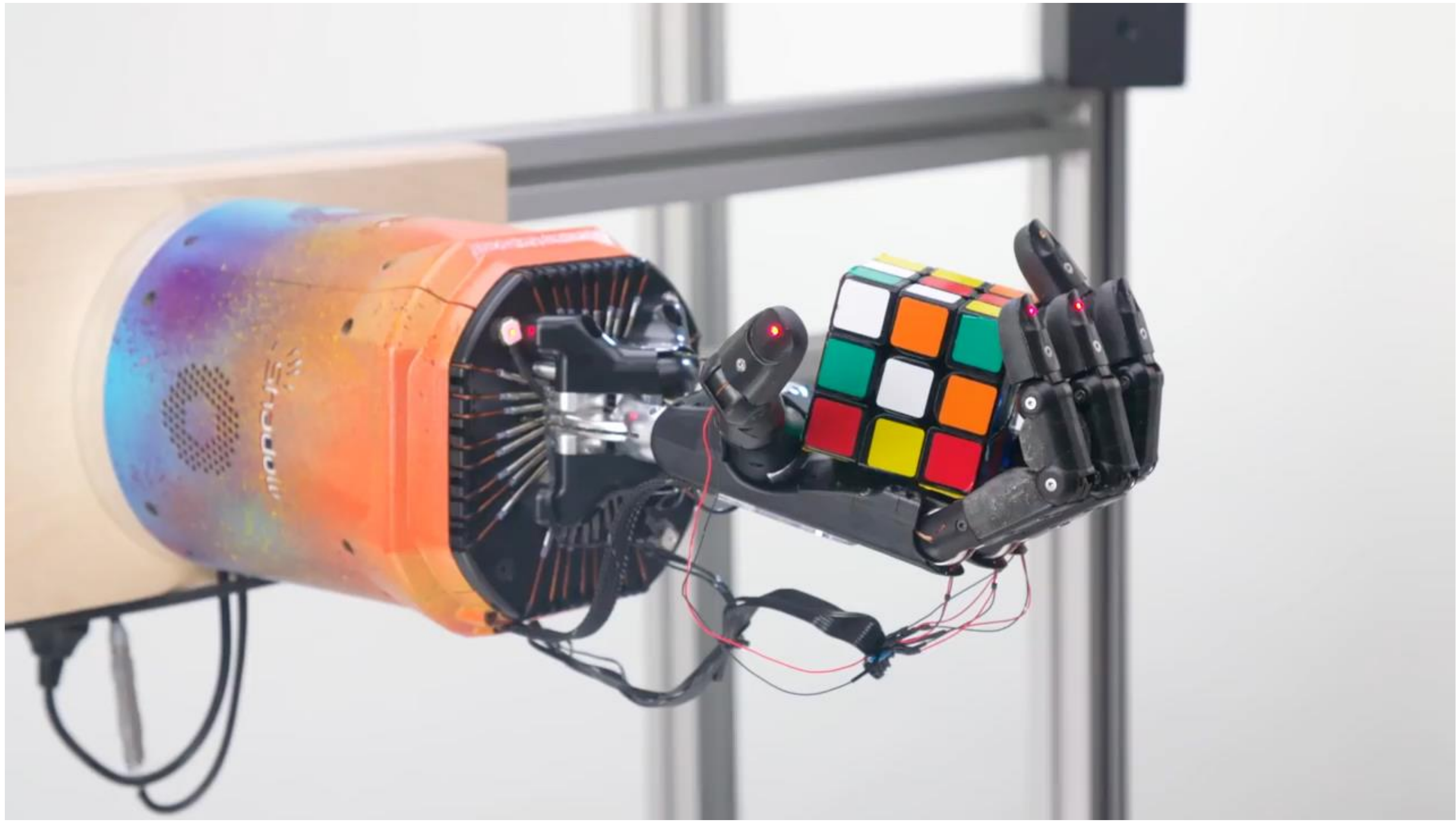
2015 **Human-level control**
[Deepmind]

AlphaGo
[Deepmind]

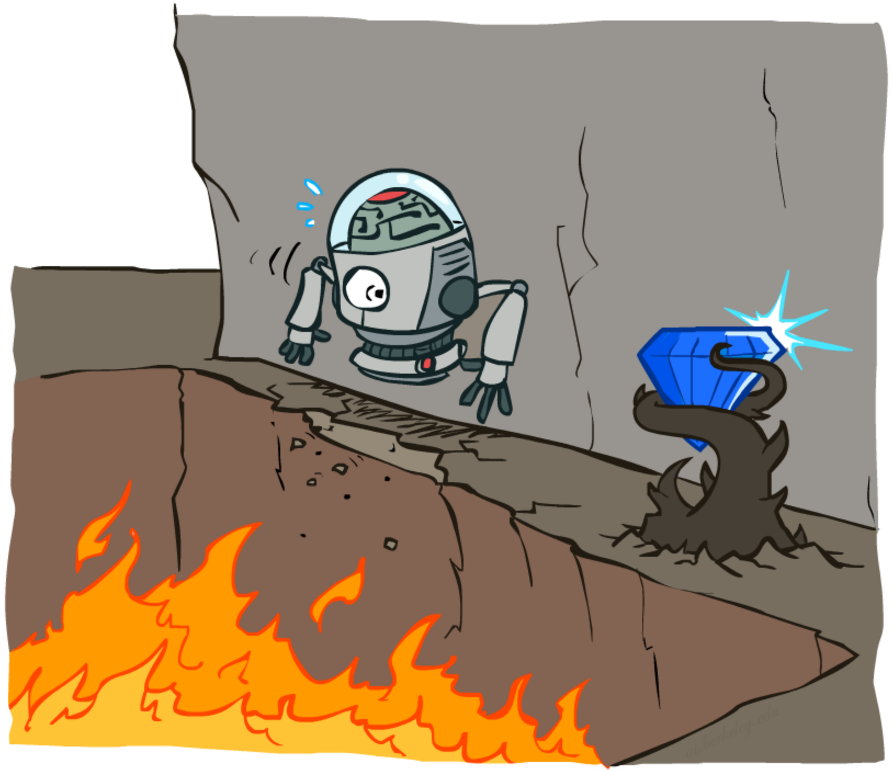
2016 **3D locomotion (TRPO+GAE)**
[Berkeley]

Real Robot Manipulation (GPS)
[Berkeley]

2019 **Rubik's Cube (PPO+DR)**
[OpenAI]

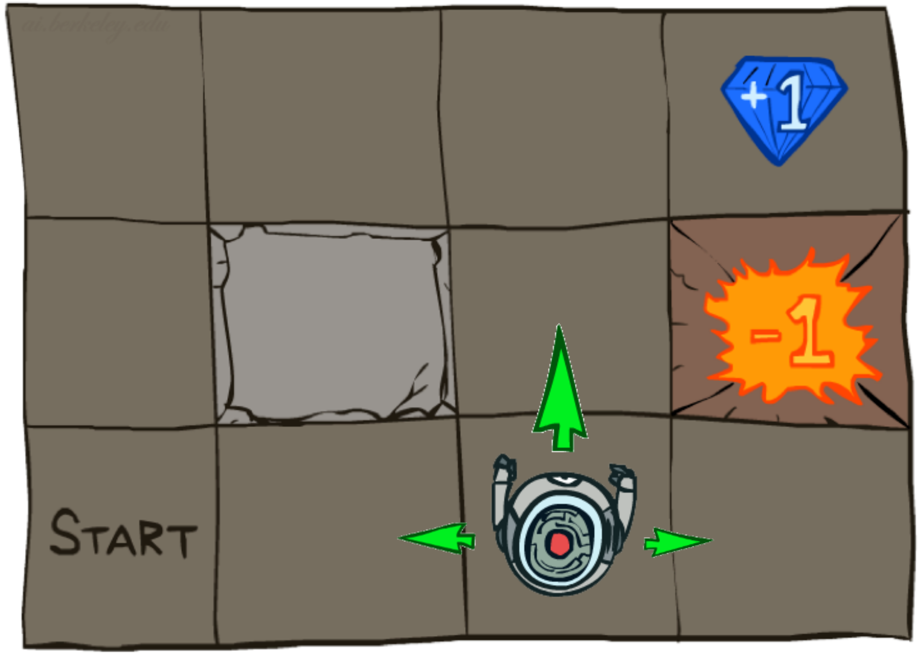


OpenAI



Non-Deterministic Search

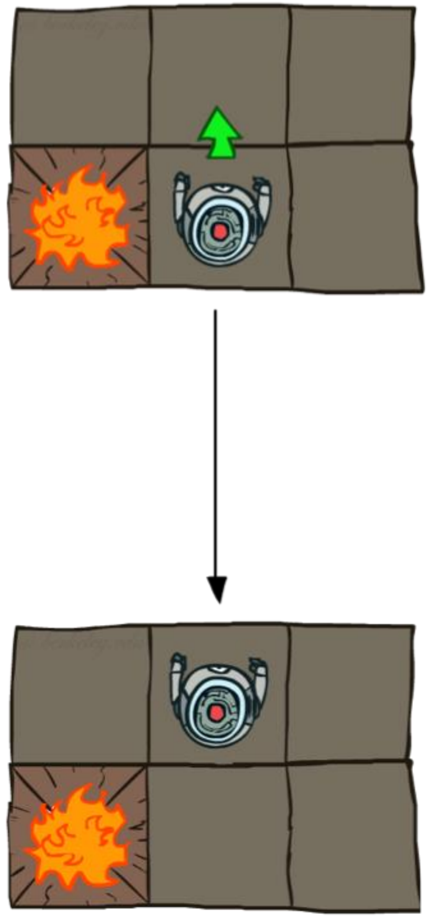
Example: Grid World



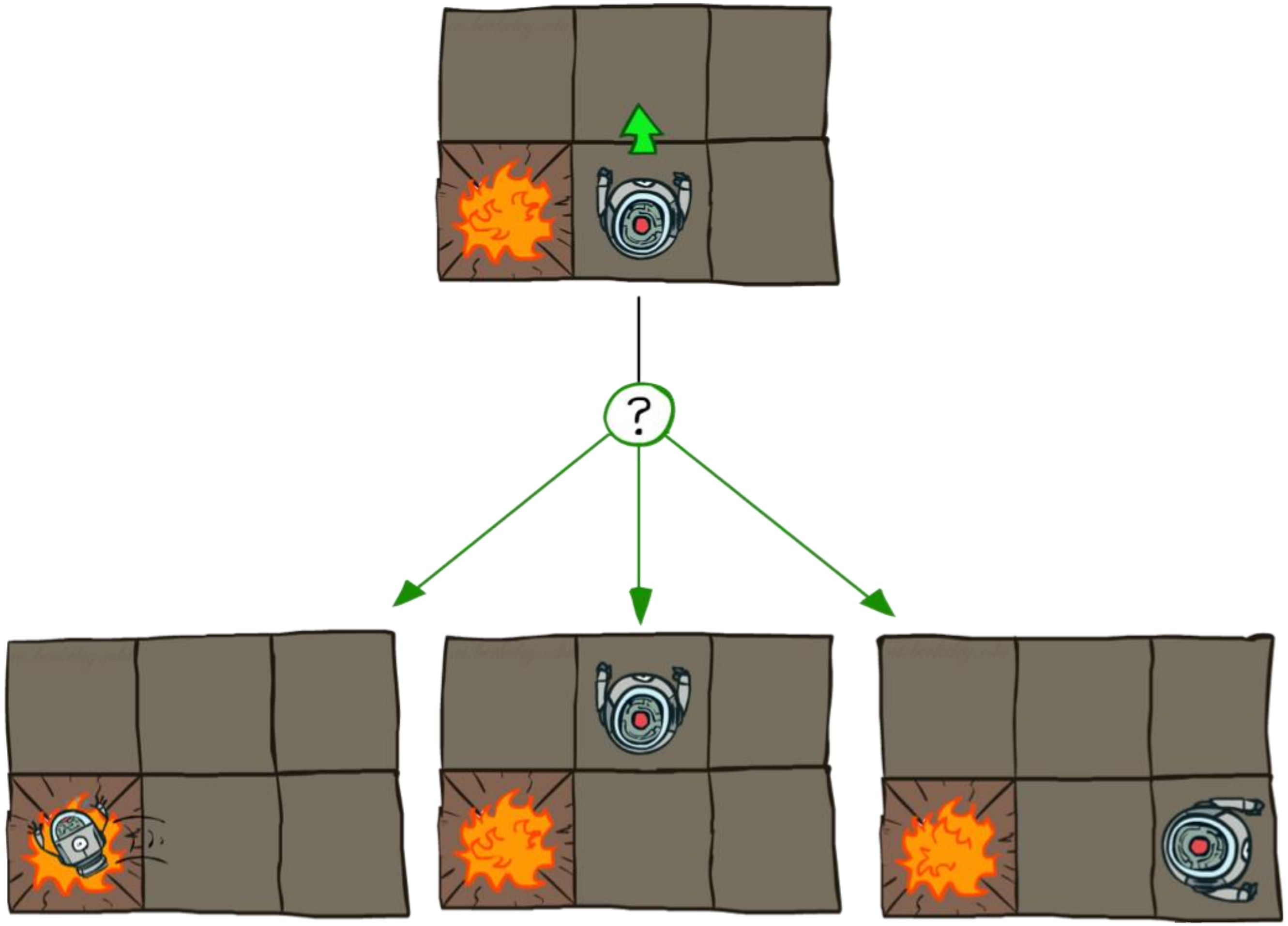
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is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



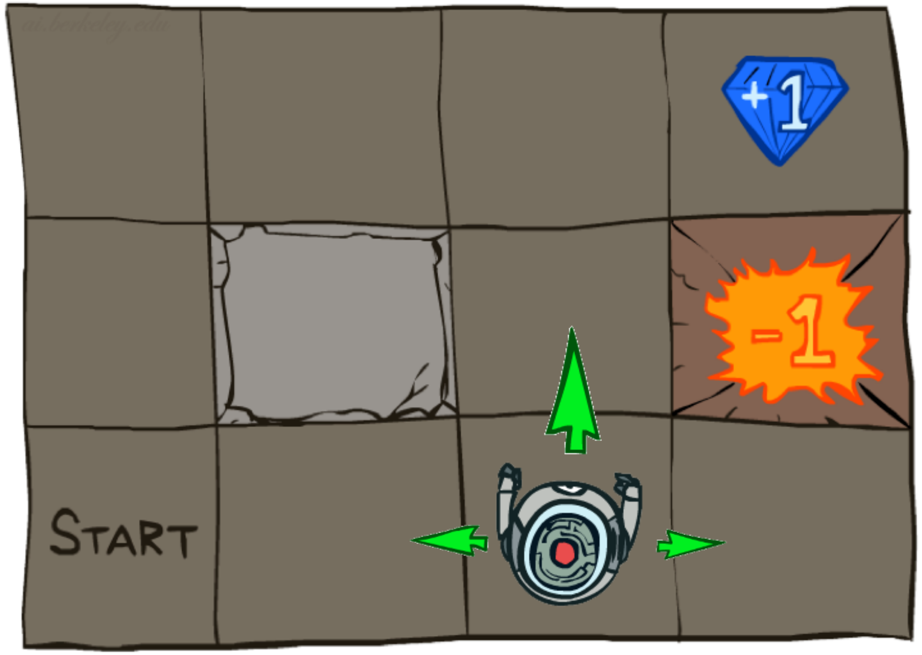
Stochastic Grid World



Markov Decision Processes

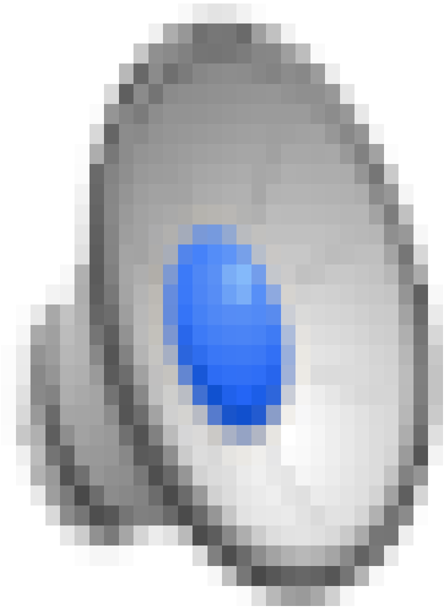
- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s' | s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
  - 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



[Demo – gridworld manual intro (L8D1)]

Video of Demo Gridworld Manual Intro



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

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

$$=$$

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

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

Andrey Markov
(1856-1922)

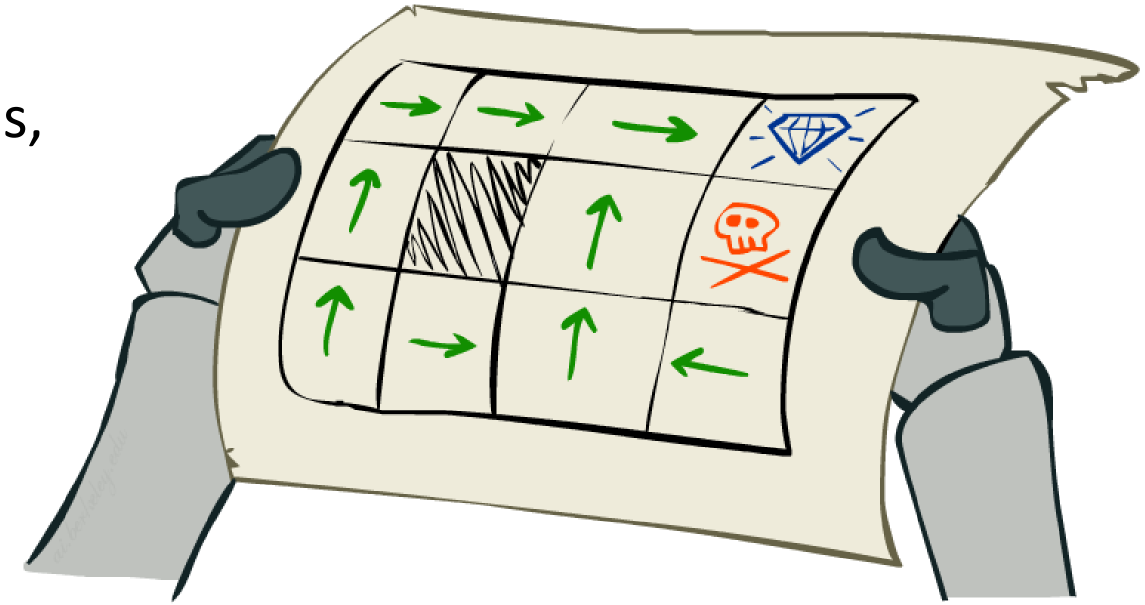
Policies

- 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

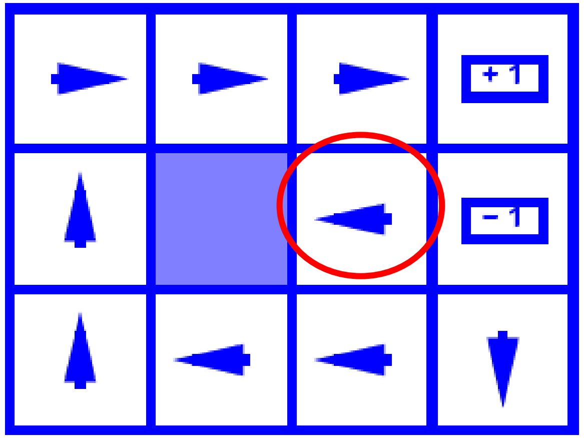
  policy $\pi^*: S \rightarrow A$

  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

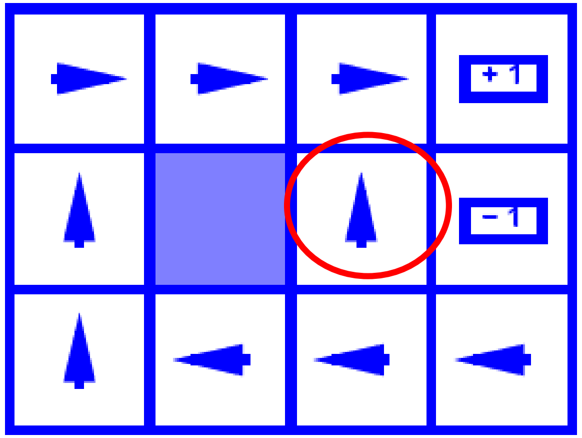


Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

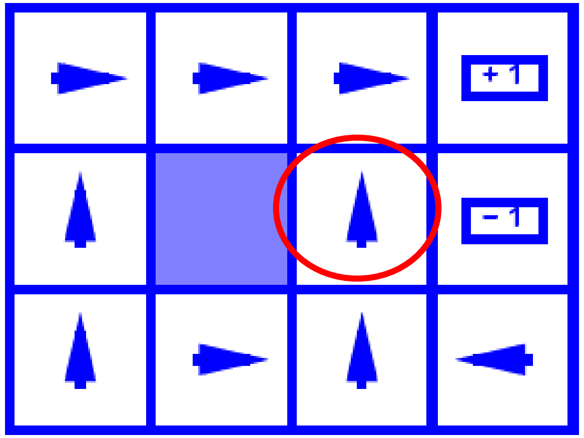
Optimal Policies



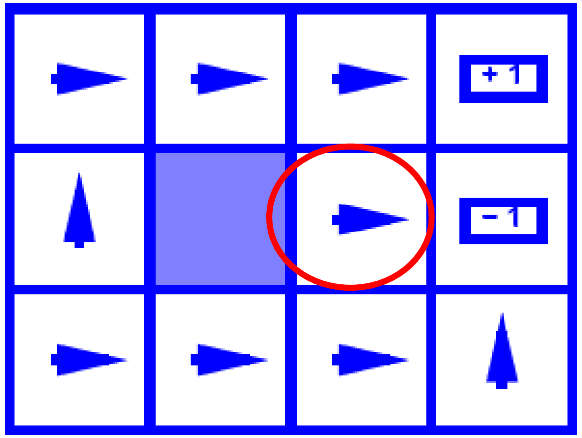
R(s) = -0.01



R(s) = -0.03

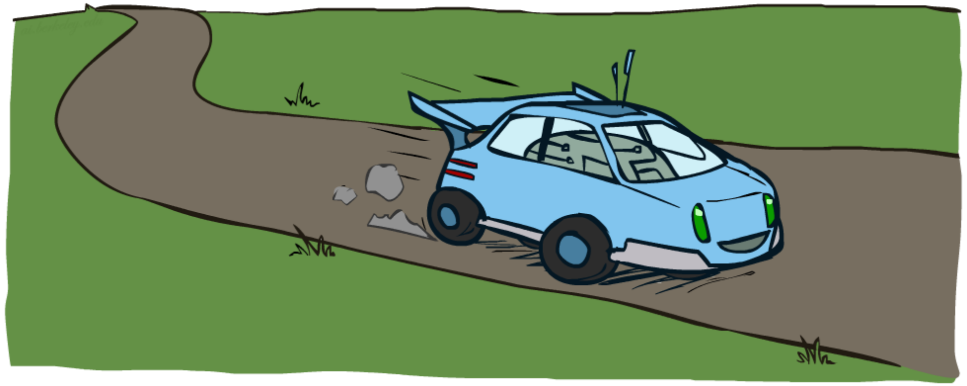


R(s) = -0.4

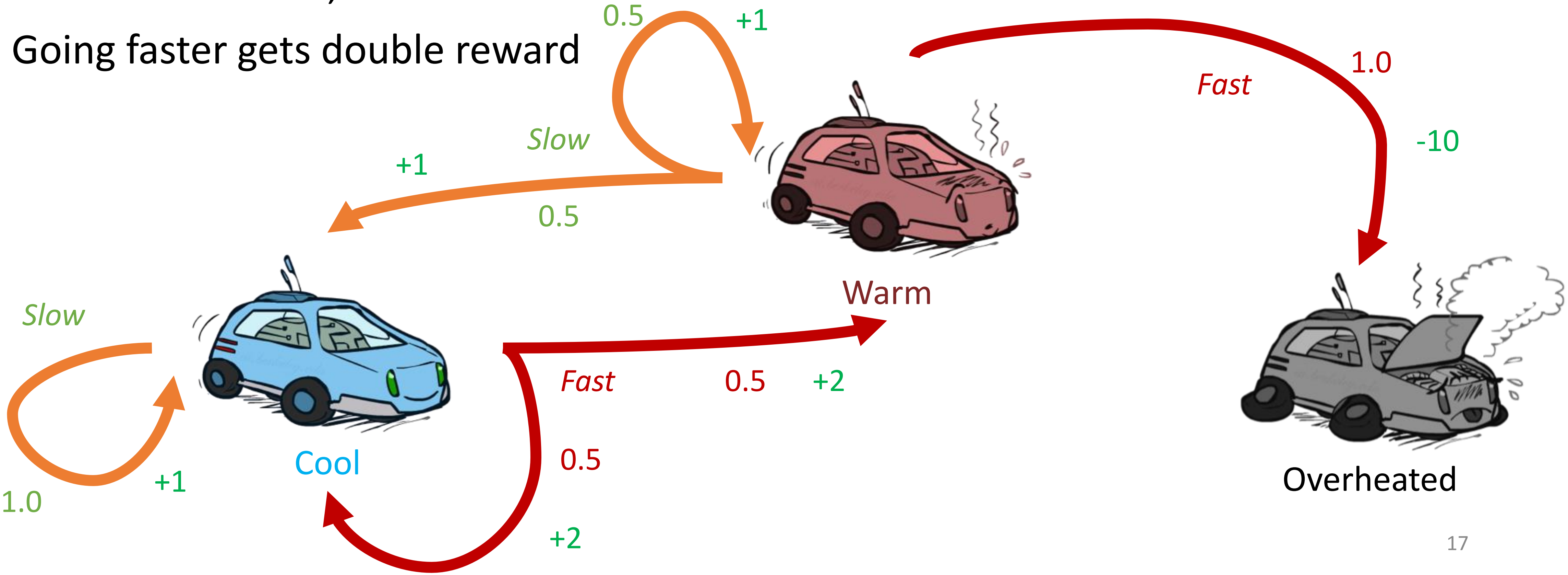


R(s) = -2.0

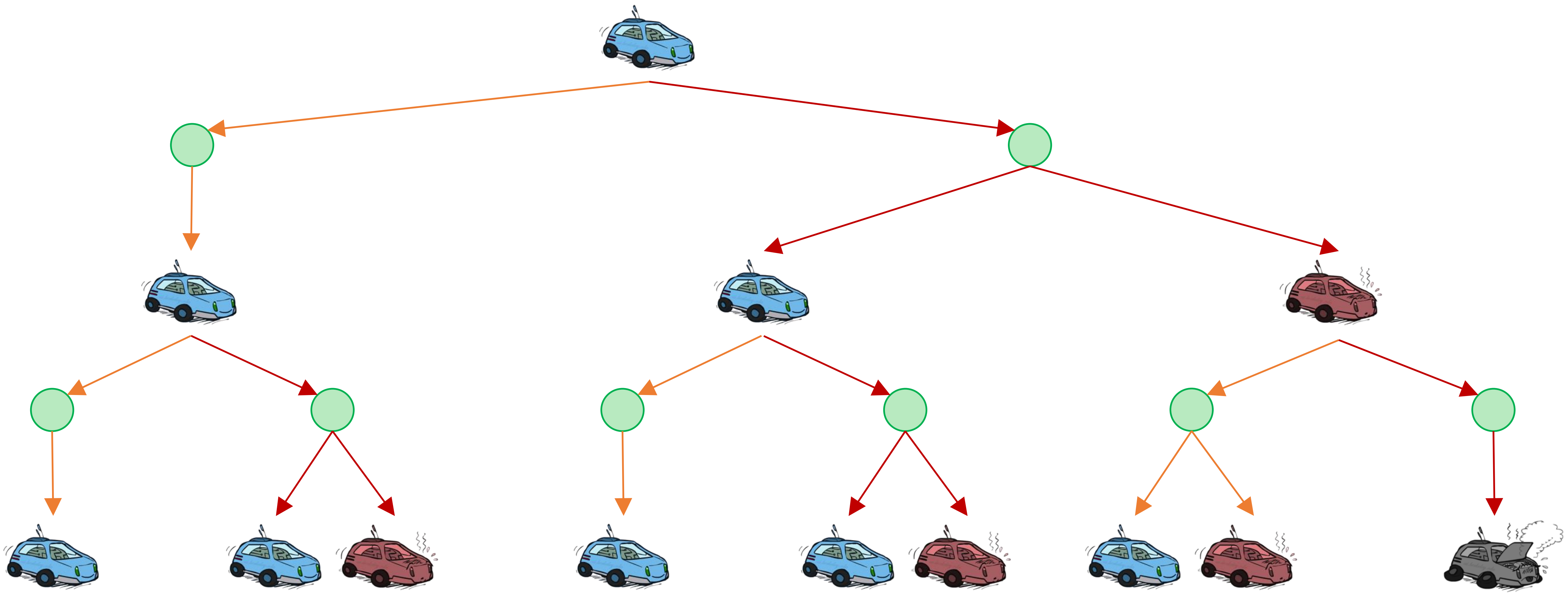
Example: Racing



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

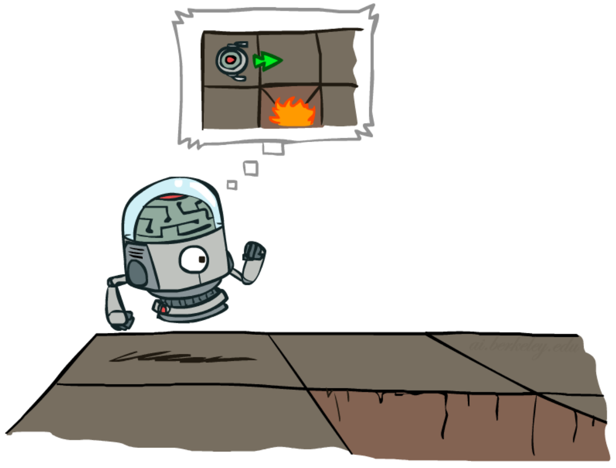


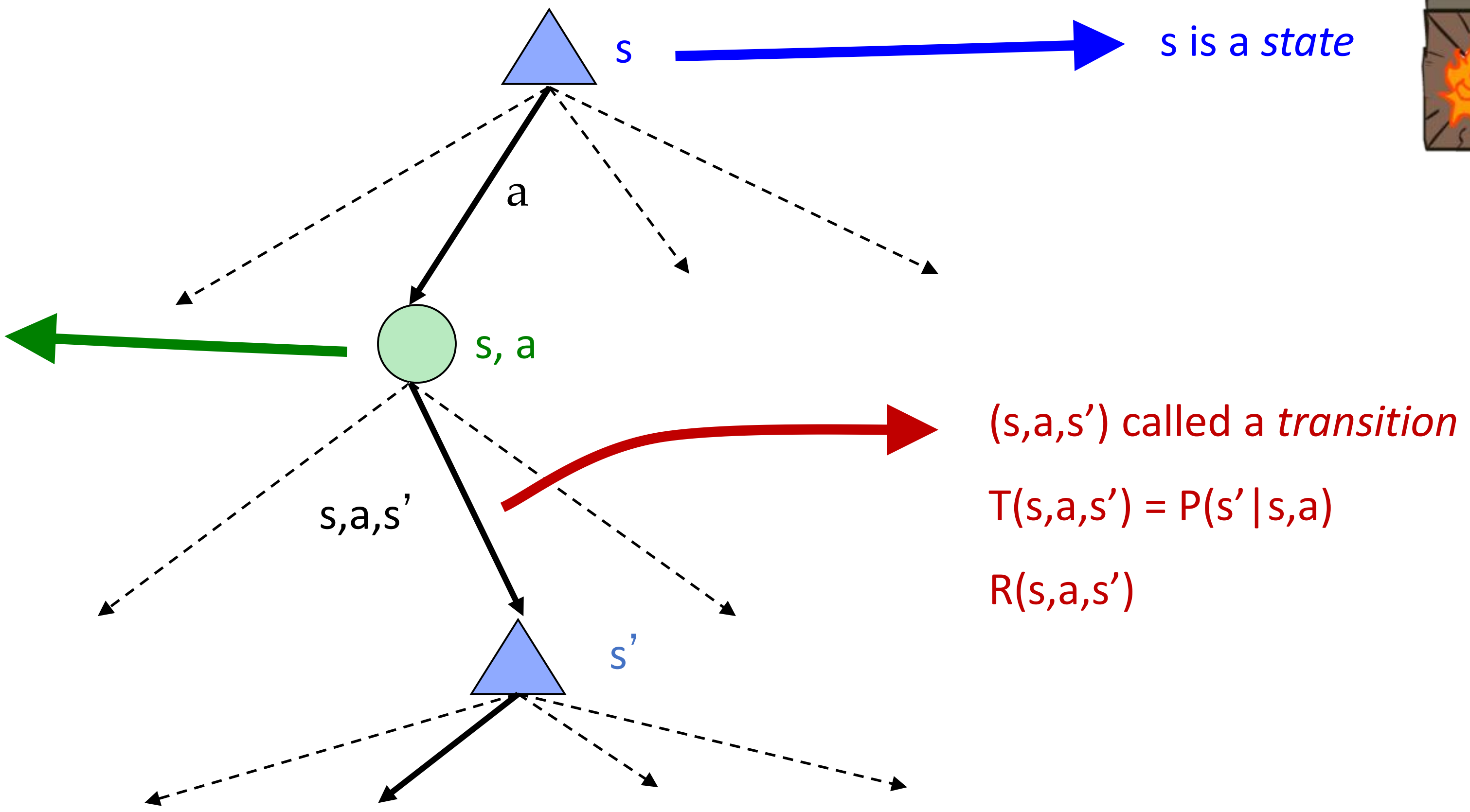
Example: Racing - Search Tree

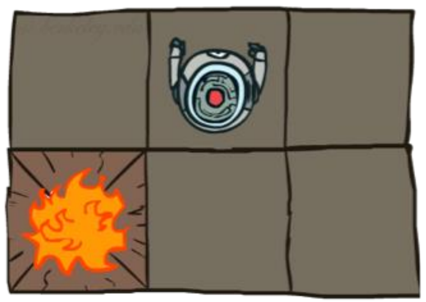


MDP Search Trees

- Each MDP state projects an expectimax-like search tree







Utilities of Sequences

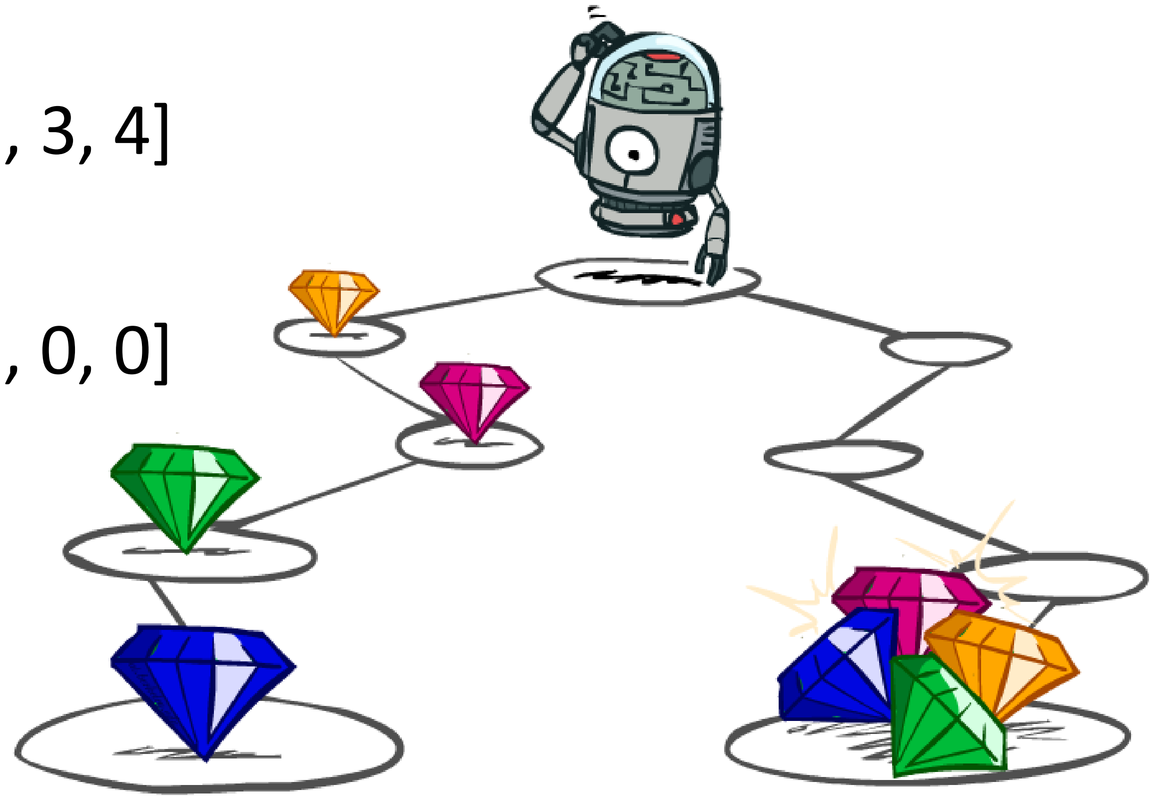
- What preferences should an agent have over reward sequences?

- More or less?

  [1, 2, 2] or [2, 3, 4]

- Now or later?

  [0, 0, 1] or [1, 0, 0]



Utilities of Sequences: 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

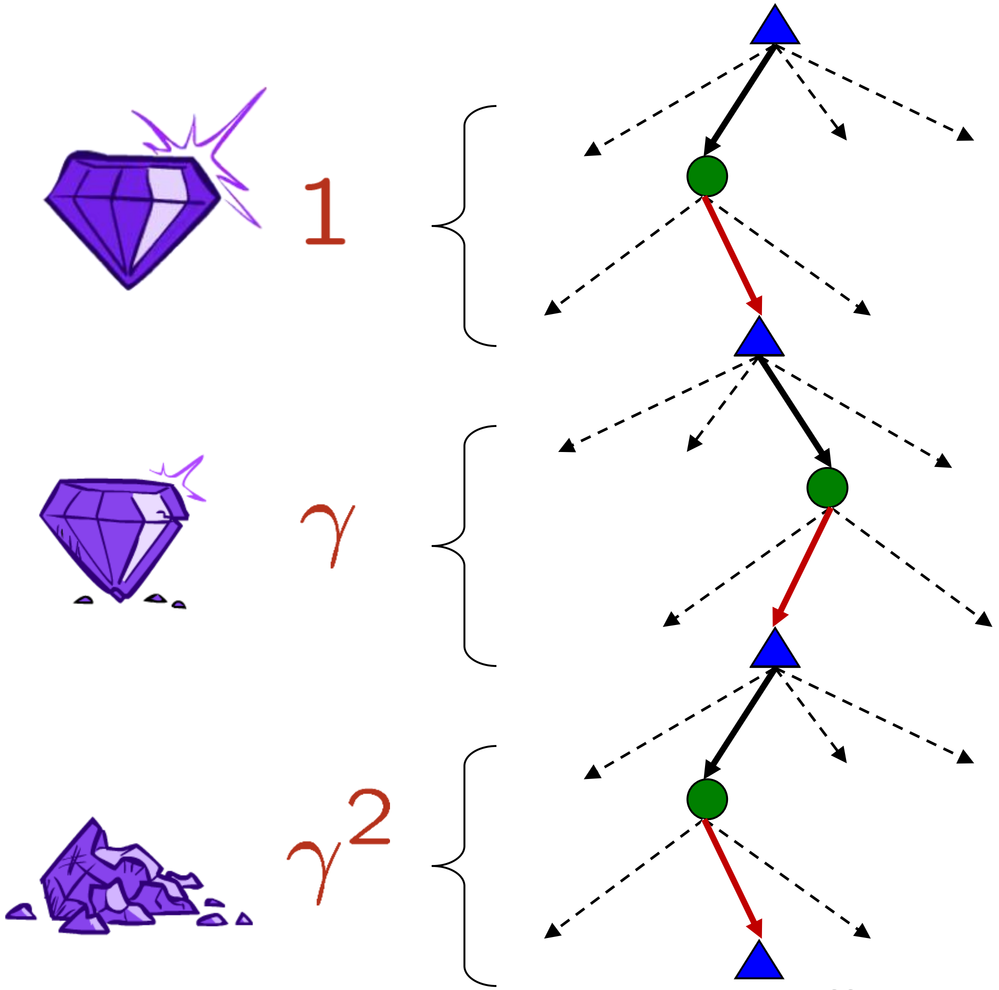
Worth Next Step

γ^2

Worth In Two Steps

Utilities of Sequences: Discounting 2

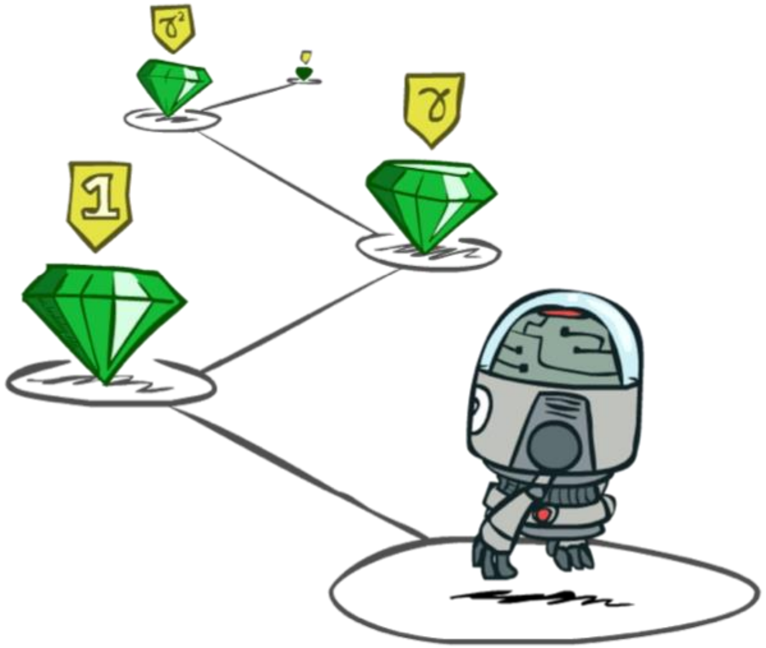
- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Reward now is better than later
  - Can also think of it as a 1-gamma chance of ending the process at every step
  - Also helps our algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



Utilities of Sequences: Stationary Preferences

- Theorem: if we assume stationary preferences:

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$
$$\Updownarrow$$
$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$



- Then: there are only two ways to define utilities
  - Additive utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$
  - Discounted utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots$

Failure of Stationary Preferences

- Can $U_\gamma + U_{\gamma'}$ define a stationary preference?

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

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$
$$\Updownarrow$$
$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$

- No!
- Example:
  - $(U_{0.9} + U_{0.5})\left(\frac{3}{4}, 0,0, \ldots\right) > (U_{0.9} + U_{0.5})(0,1,0, \ldots)$
  - $(U_{0.9} + U_{0.5})\left(r, \frac{3}{4}, 0, \ldots\right) < (U_{0.9} + U_{0.5})(r, 0,1,0,0, \ldots)$

Proof Sketch

- Theorem (F. Riesz) For any inner product space H, let f be a continuous linear functional, that is $f: H \to \mathbb{R}$ is continuous and satisfies $f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$. Then f can be written as
$$f(x) =< z, x >$$
for some $z \in H$

- Then by $a_0 + \gamma_1 a_1 > b_0 + \gamma_1 b_1 \Leftrightarrow \gamma_1 a_0 + \gamma_2 a_1 > \gamma_1 b_0 + \gamma_2 b_1$ which says $(a_0 - b_0) + \gamma_1(a_1 - b_1) > 0 \Leftrightarrow \gamma_1(a_0 - b_0) + \gamma_2(a_1 - b_1) > 0$
Then there must have $\gamma_2 = \gamma_1^2$
- Similarly for the rest

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 γ = 1, what is the optimal policy?

10	<-	<-	<-	1

- Quiz 2: For γ = 0.1, what is the optimal policy?

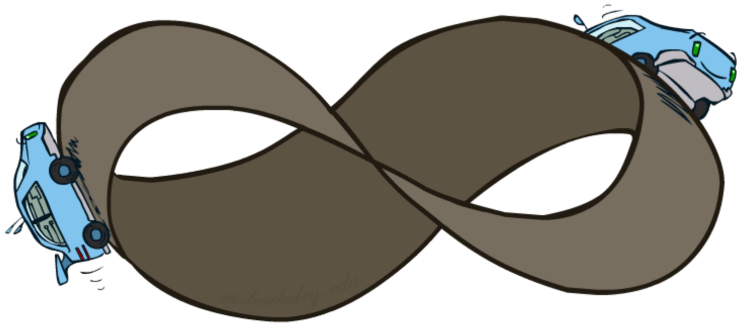
10	<-	<-	->	1

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

$1\gamma = 10\,\gamma^3$

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)
    - Gives nonstationary policies (π depends on time left)



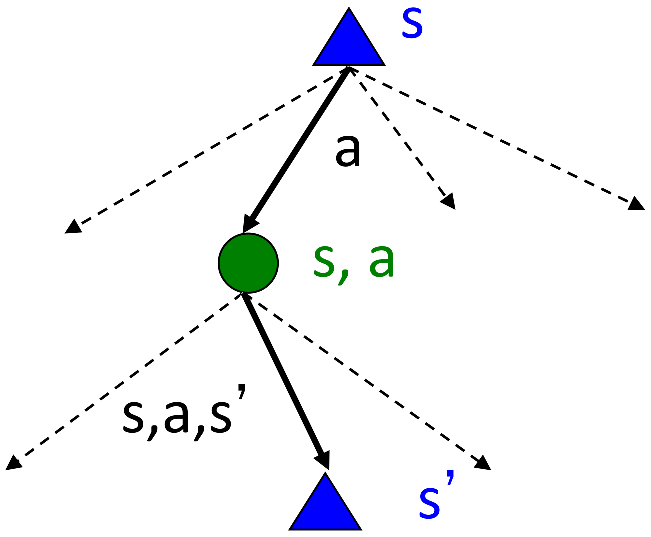
  - Discounting: use $0 < \gamma < 1$

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

    - 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)

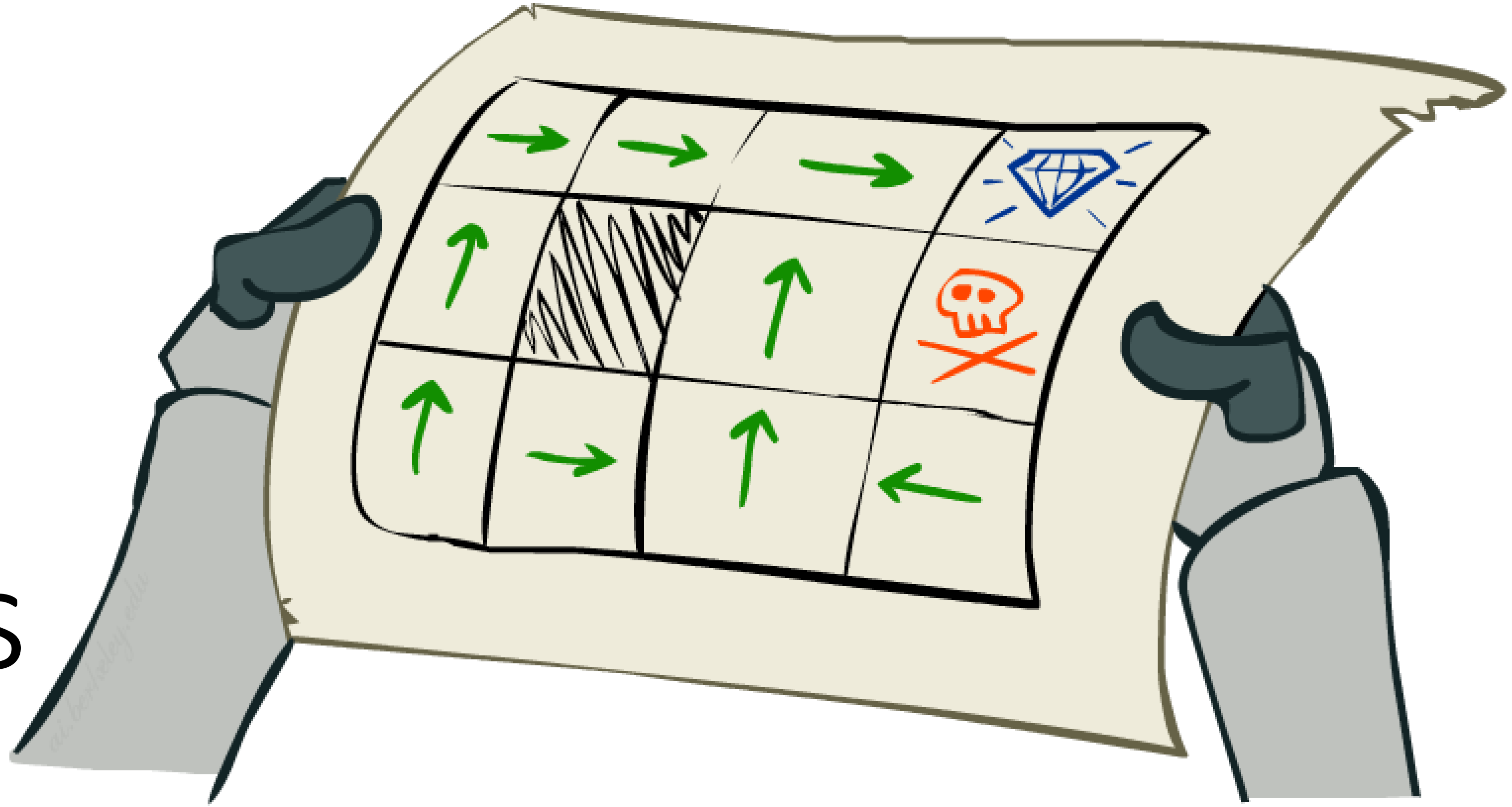
Recap: Defining MDPs

- 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)

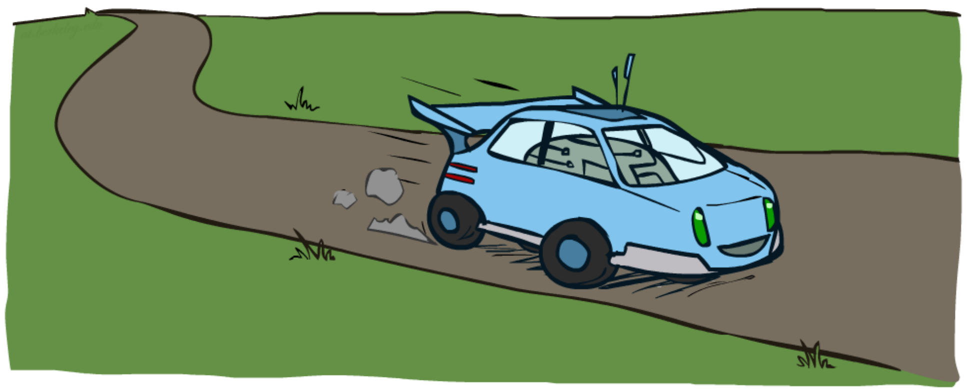


- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

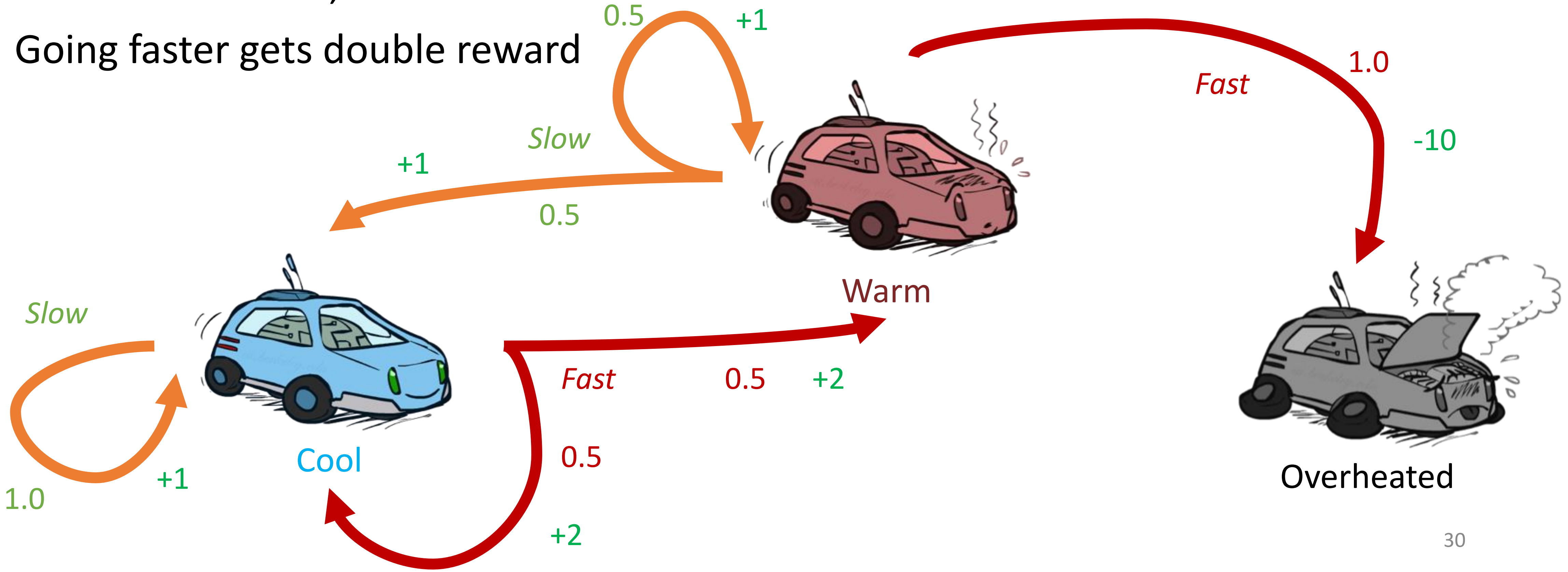
Solving MDPs



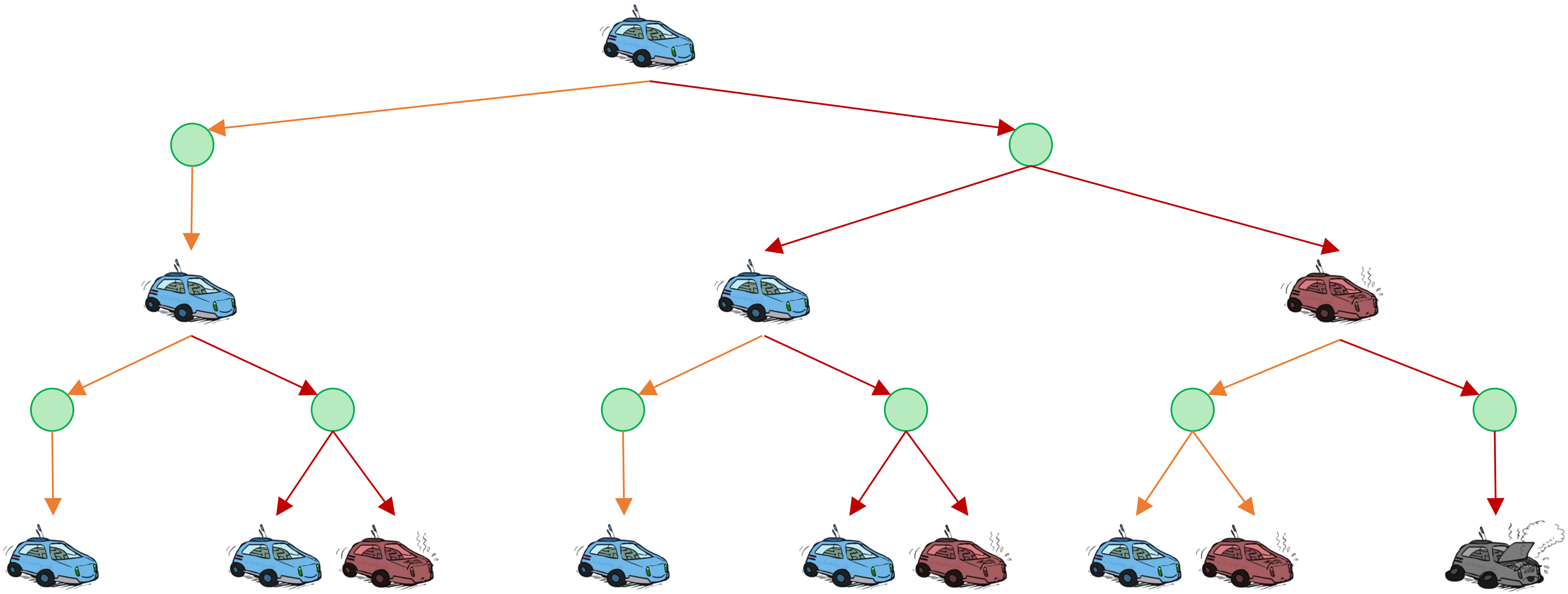
Recall: Racing MDP



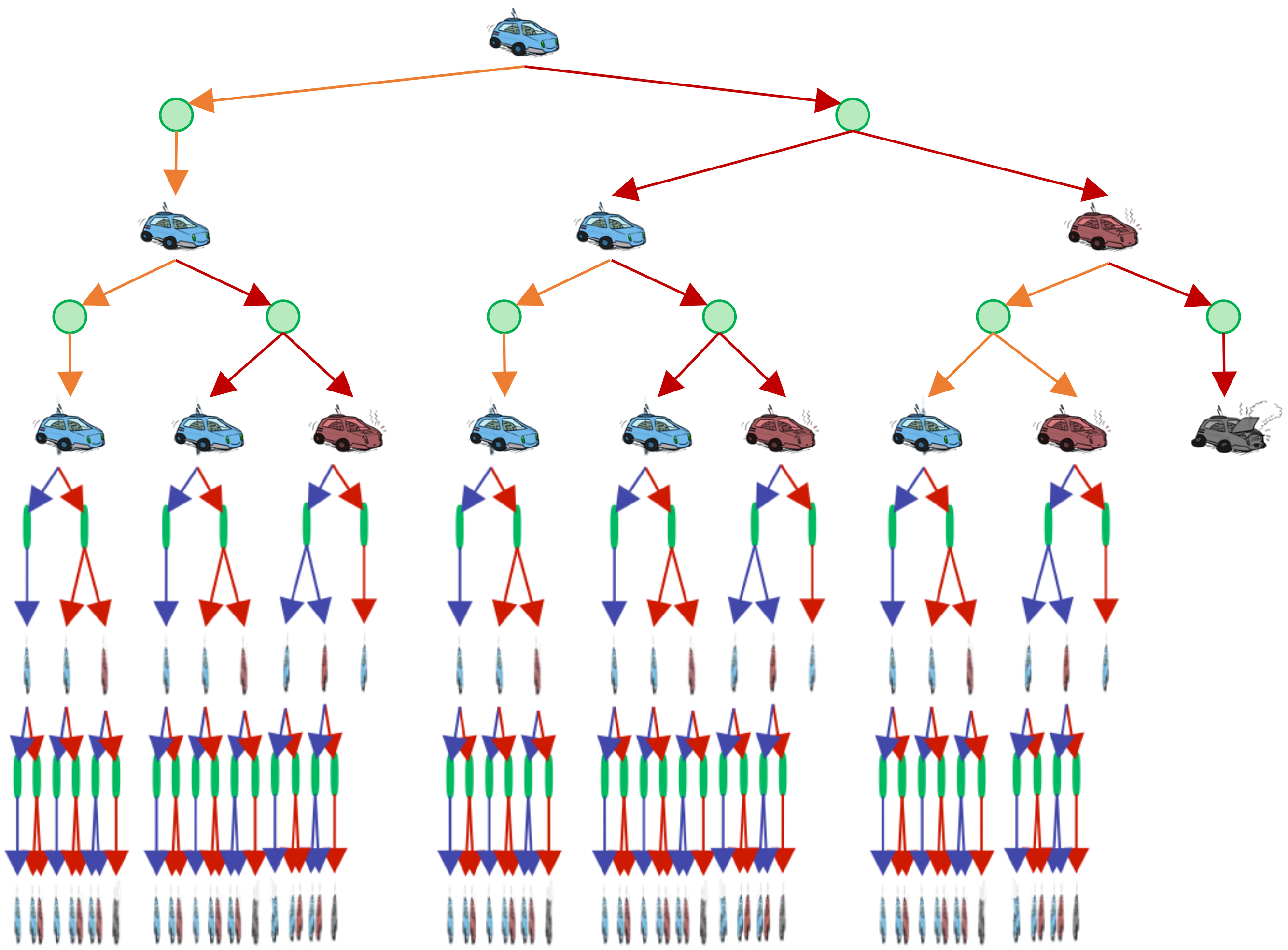
- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward



Racing Search Tree



Racing Search Tree 2

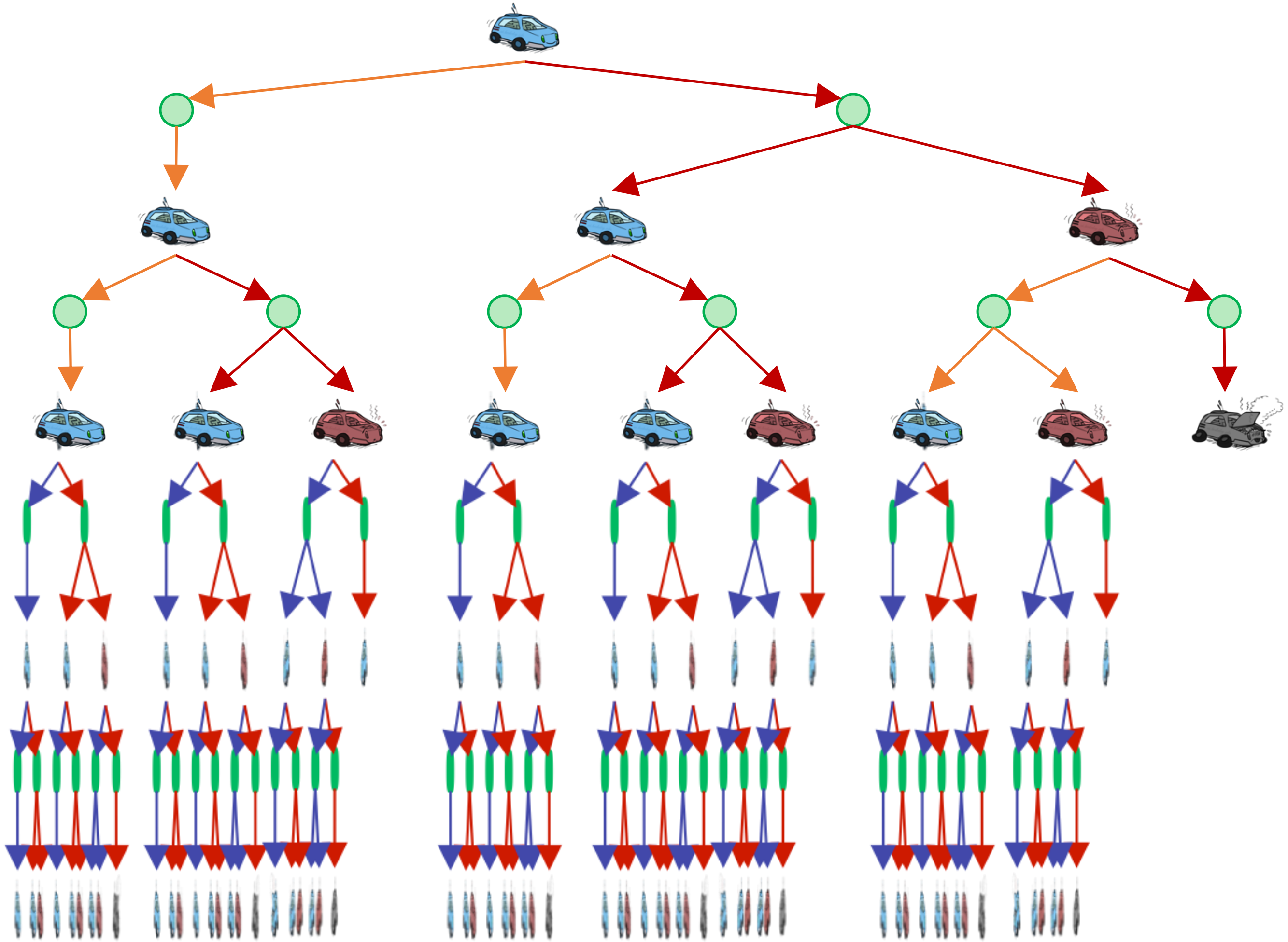


Racing Search Tree 3

- We're doing way too much work with expectimax!

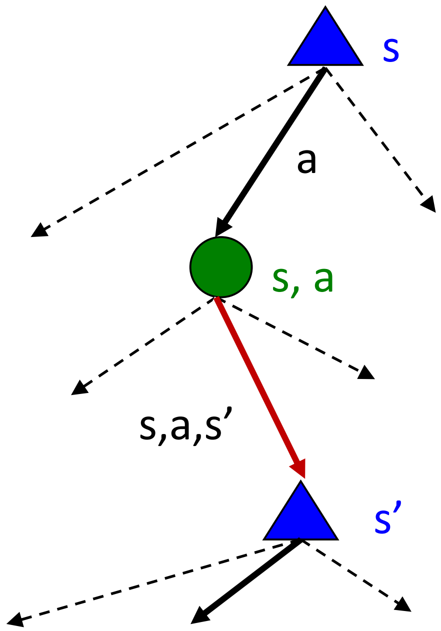
- Problem: States are repeated
  - Idea: Only compute needed quantities once

- 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 $\gamma < 1$



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:
  - π*(s) = optimal action from state s

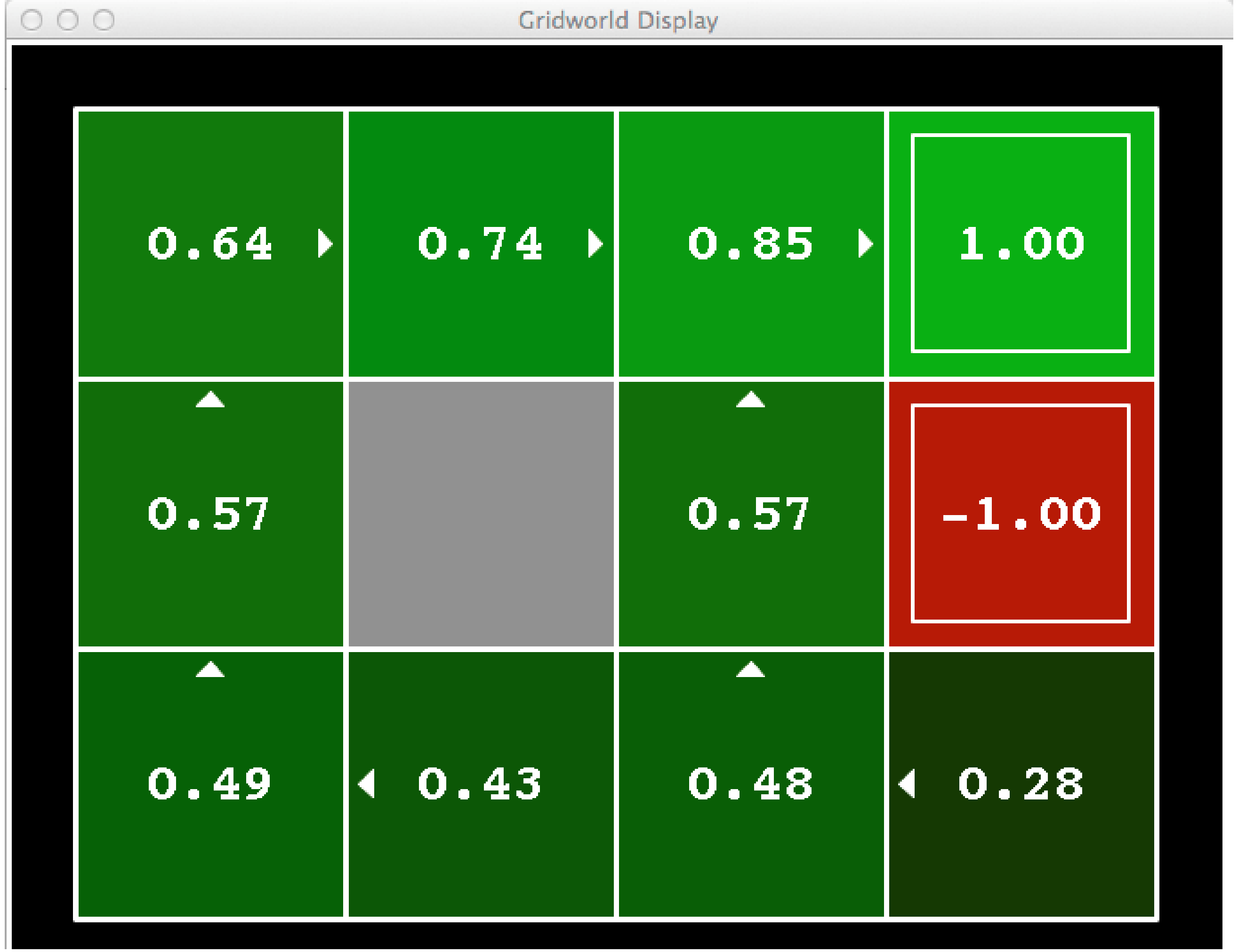


s is a *state*

(s, a) is a *q-state*

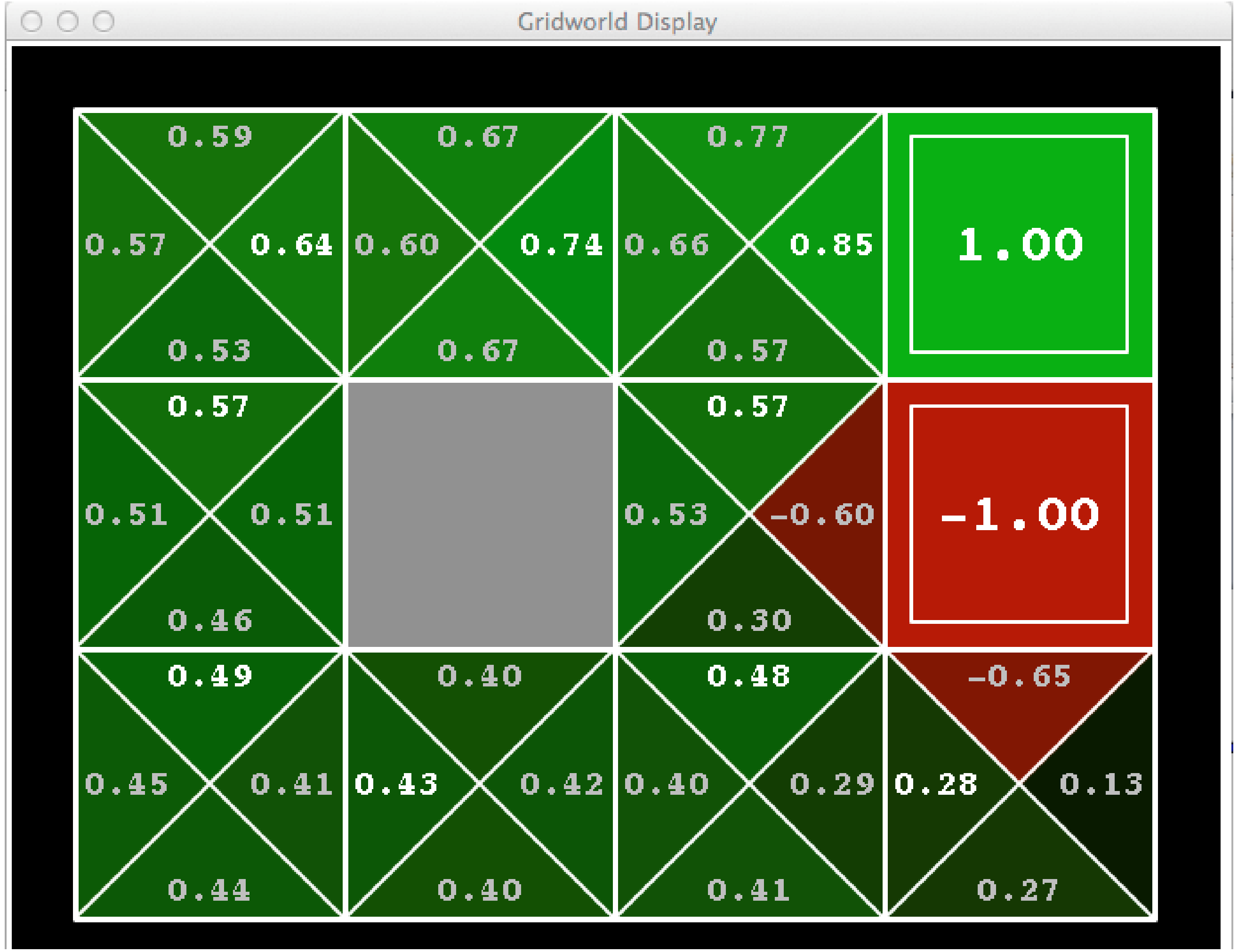
(s,a,s') is a *transition*

Gridworld V* Values



Noise = 0.2
Discount = 0.9
Living reward = 0

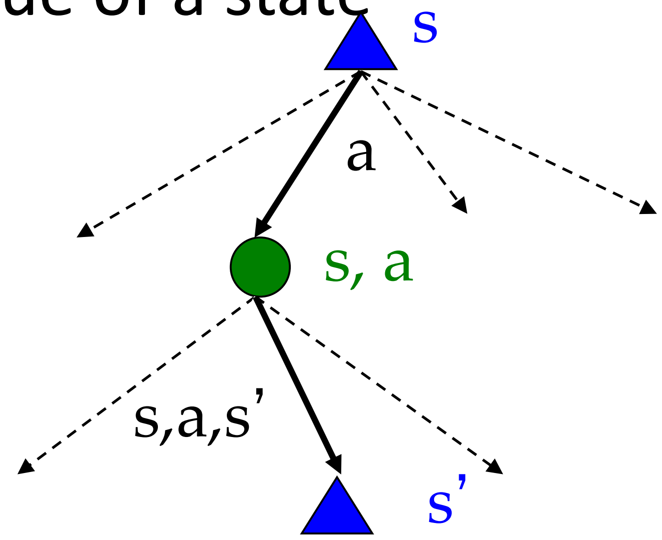
Gridworld Q* Values



Noise = 0.2
Discount = 0.9
Living reward = 0

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!
- Recursive definition of value:

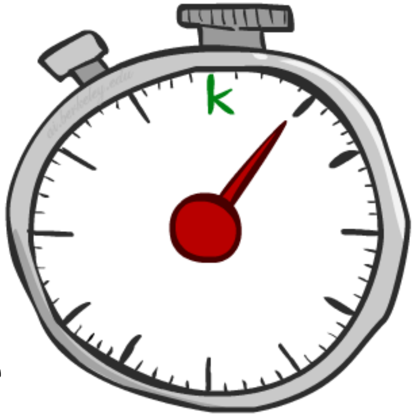


$$V^*(s) = \max_a Q^*(s,a)$$

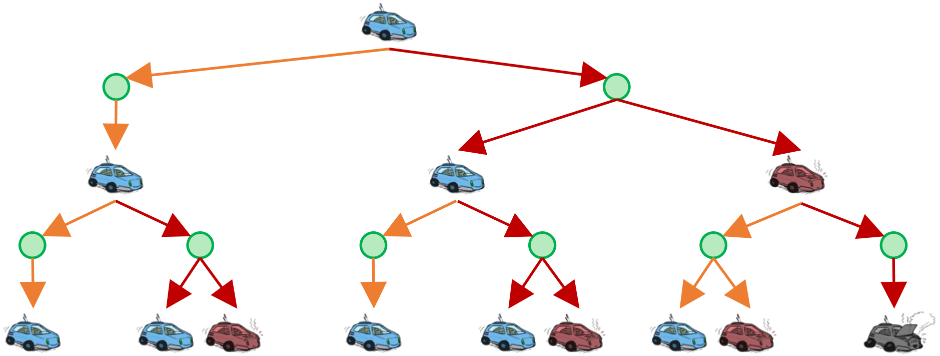
$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s') \right]$$

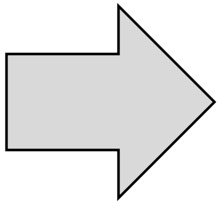
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Time-Limited Values

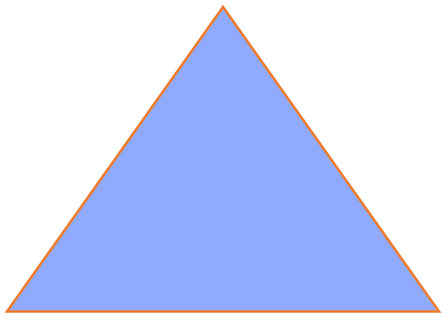


- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
  - Equivalently, it's what a depth-k expectimax would give from s



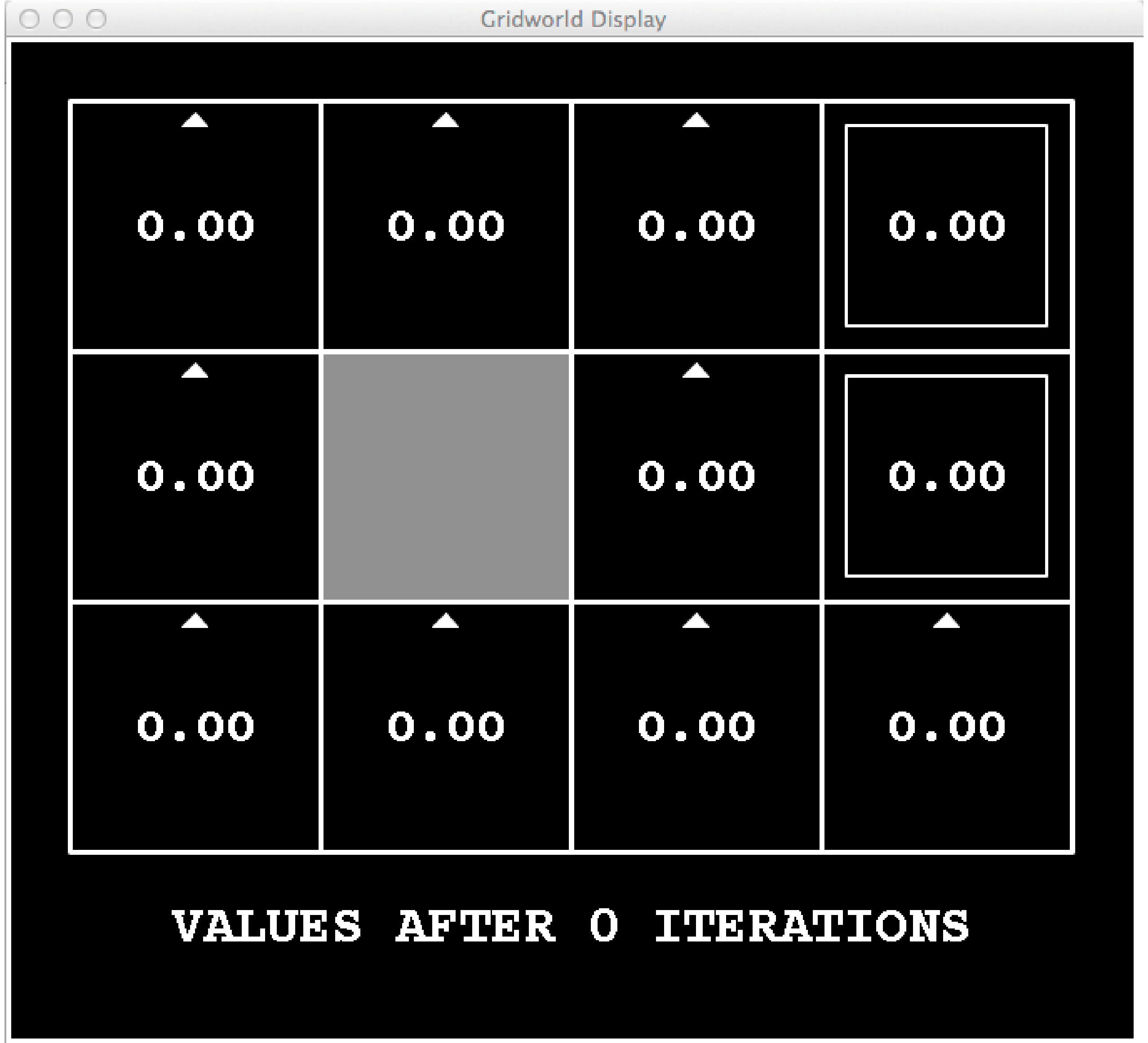


$V_2($ $)$



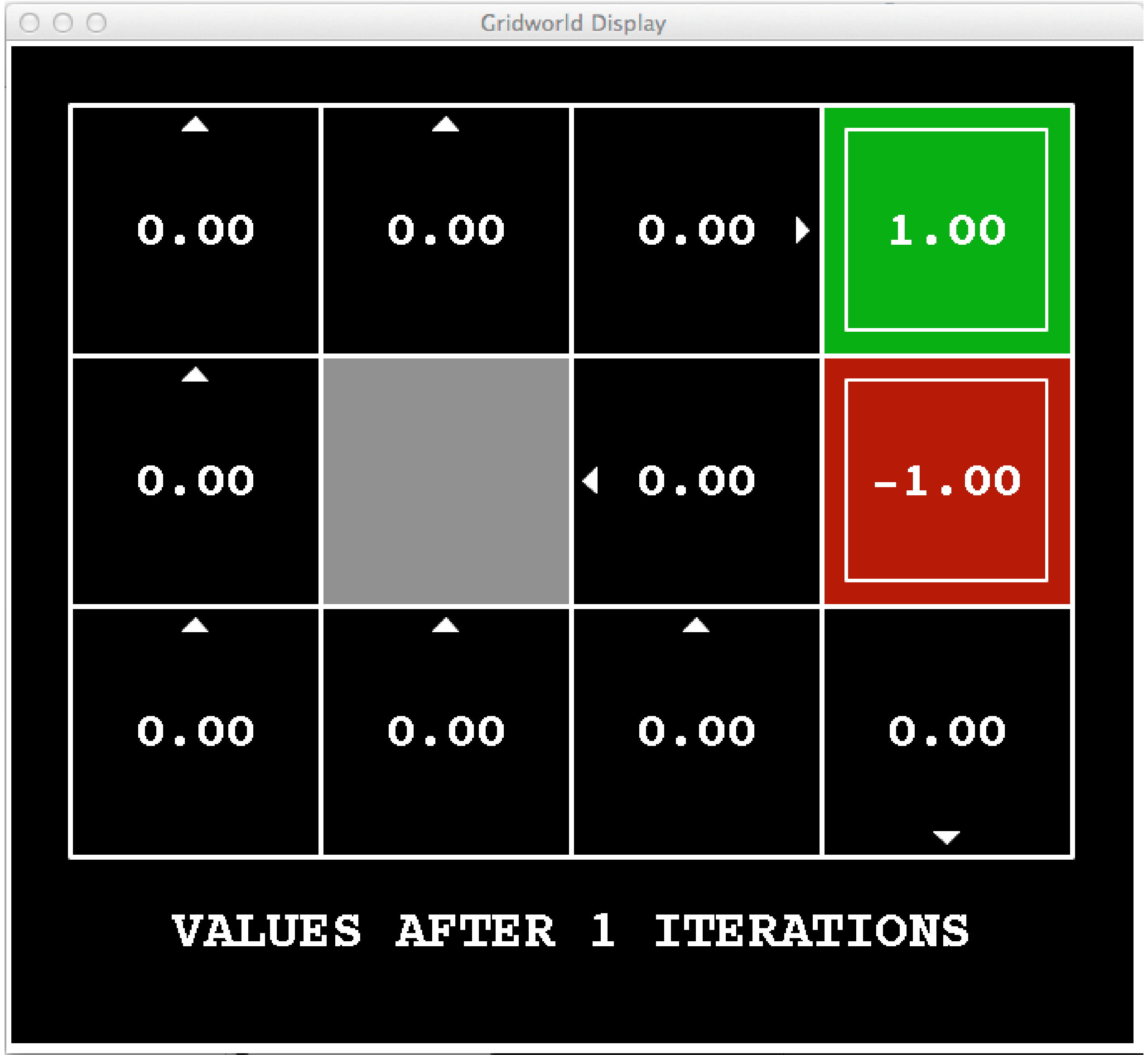
[Demo – time-limited values (L8D4)]

Gridworld: k=0



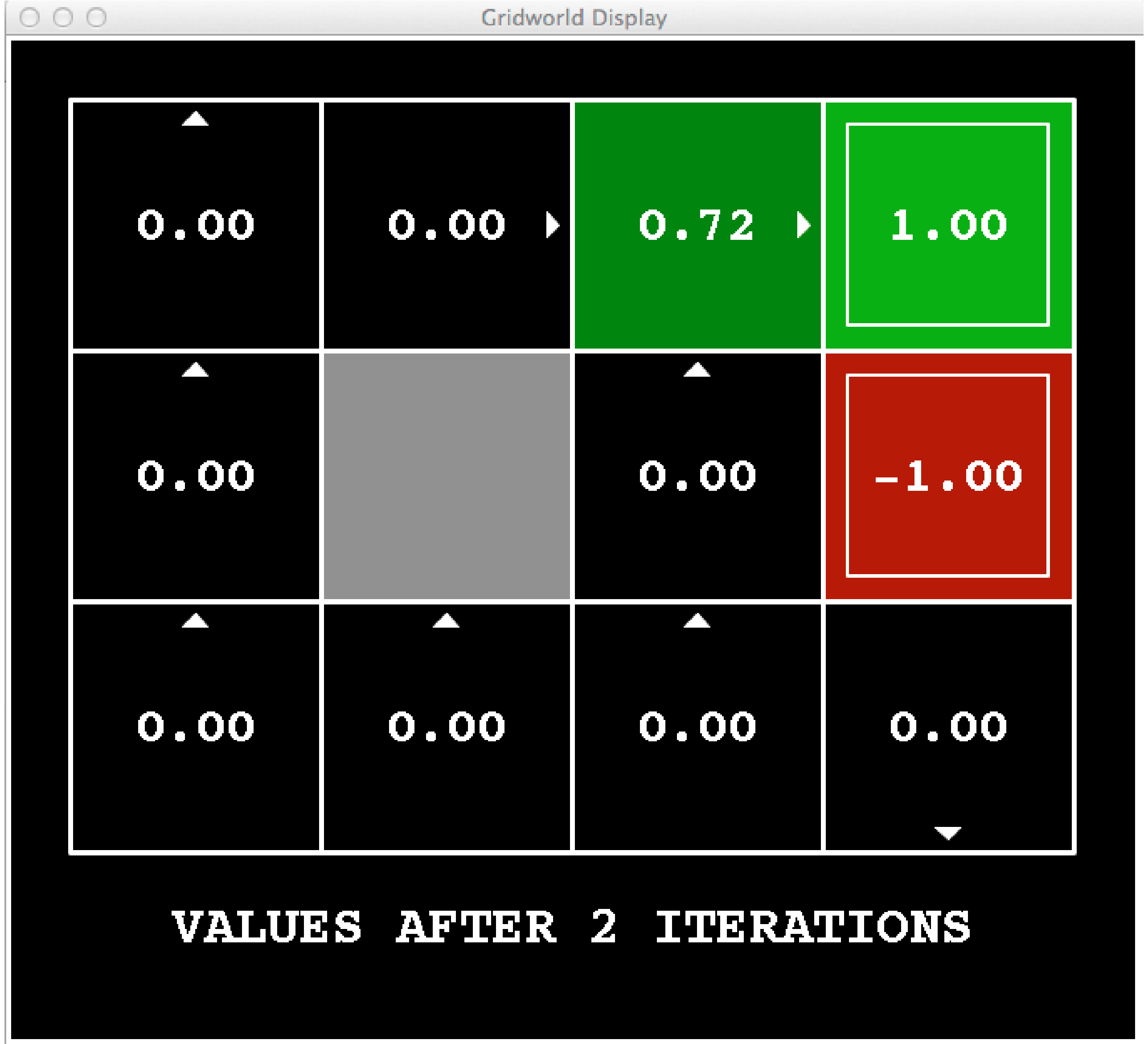
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=2

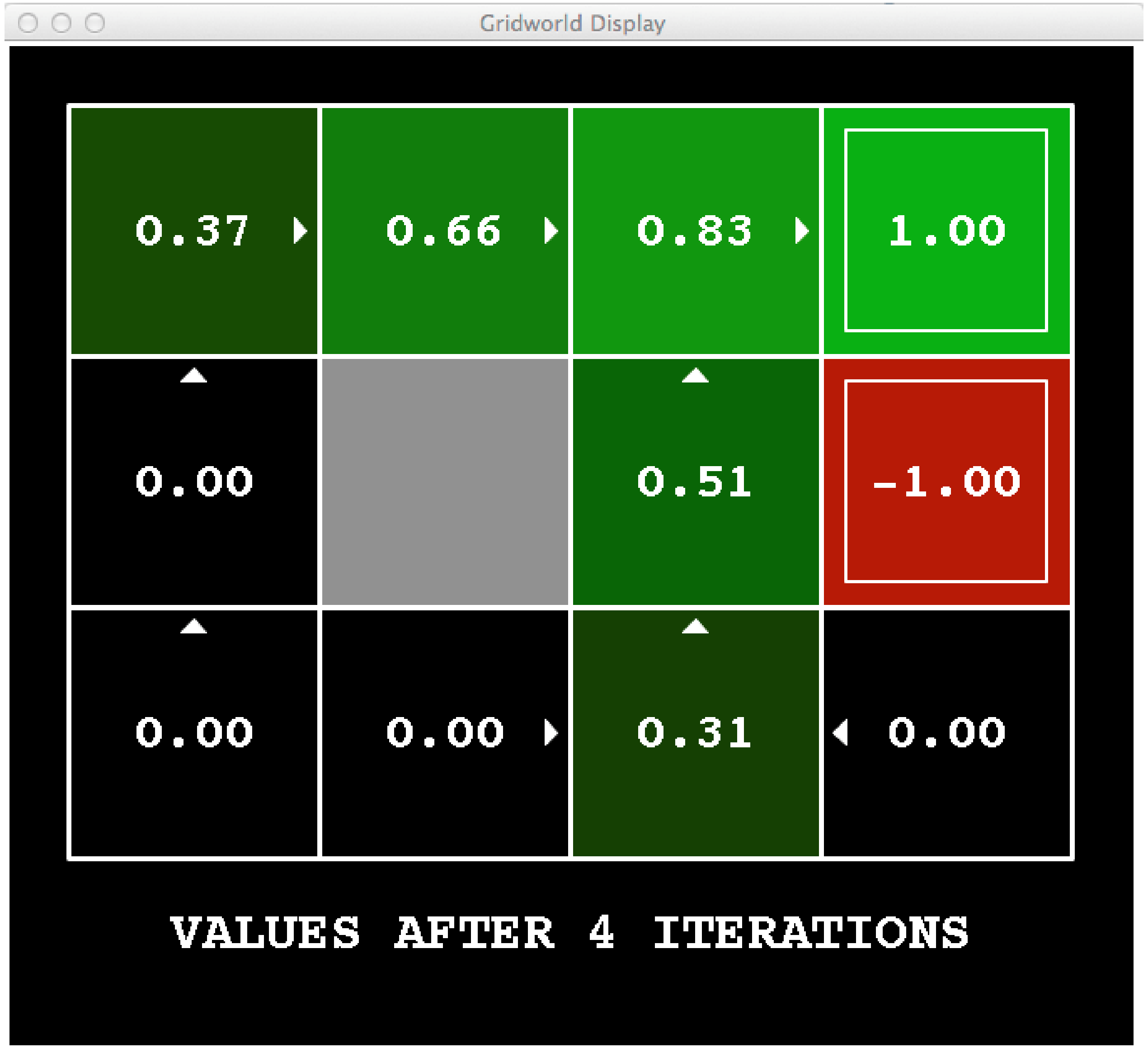


Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=3

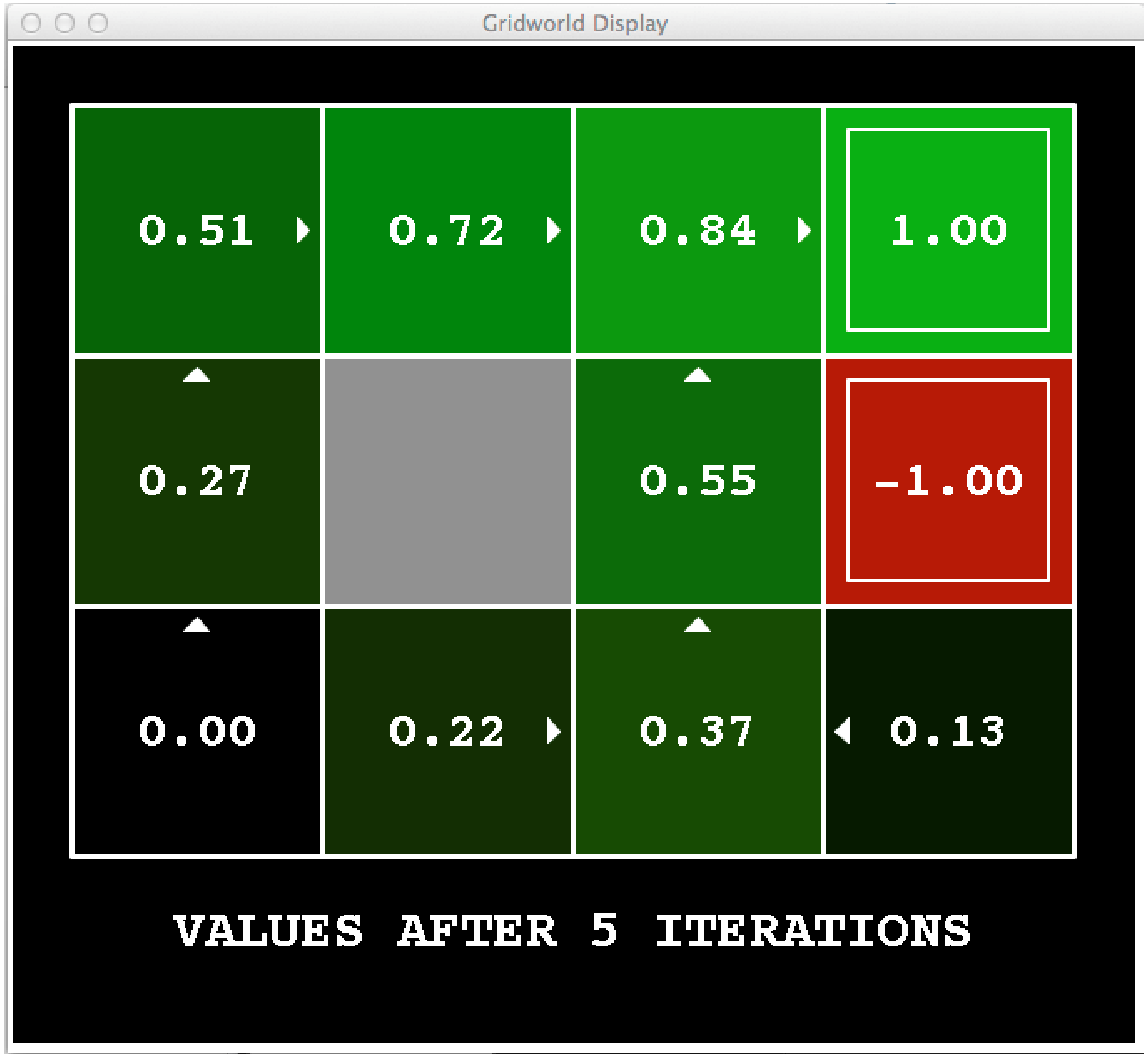
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=4



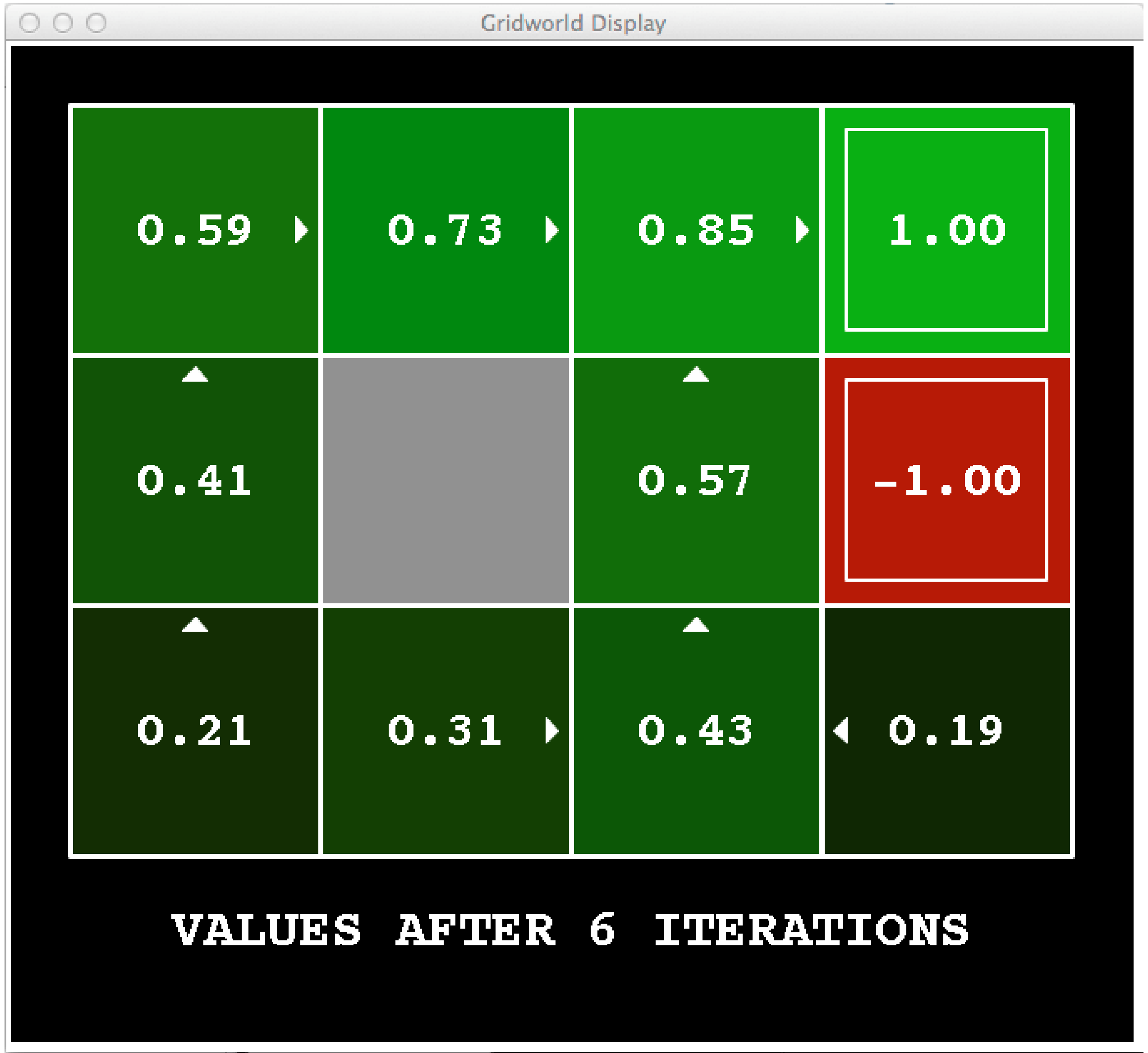
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=5



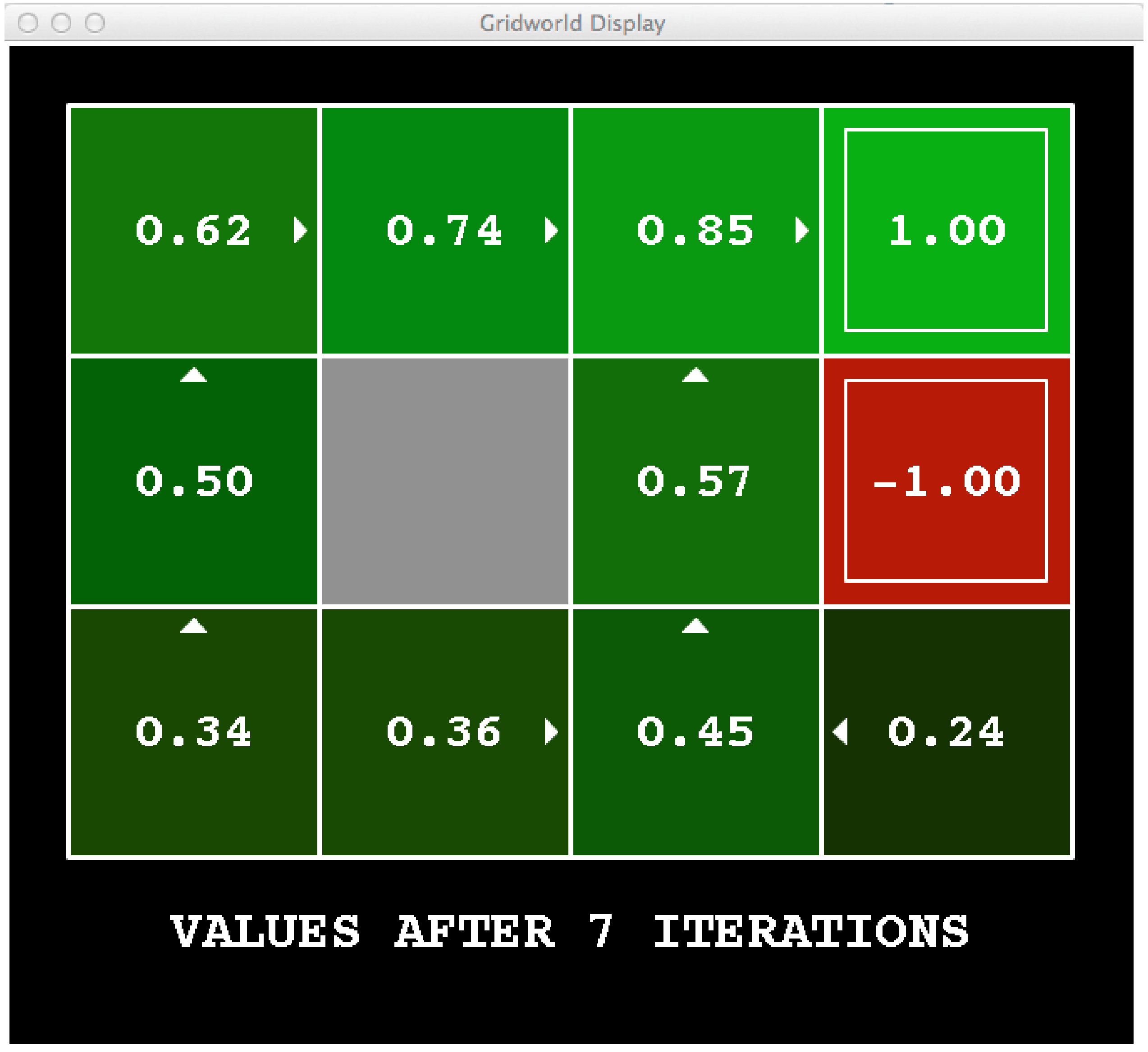
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=6



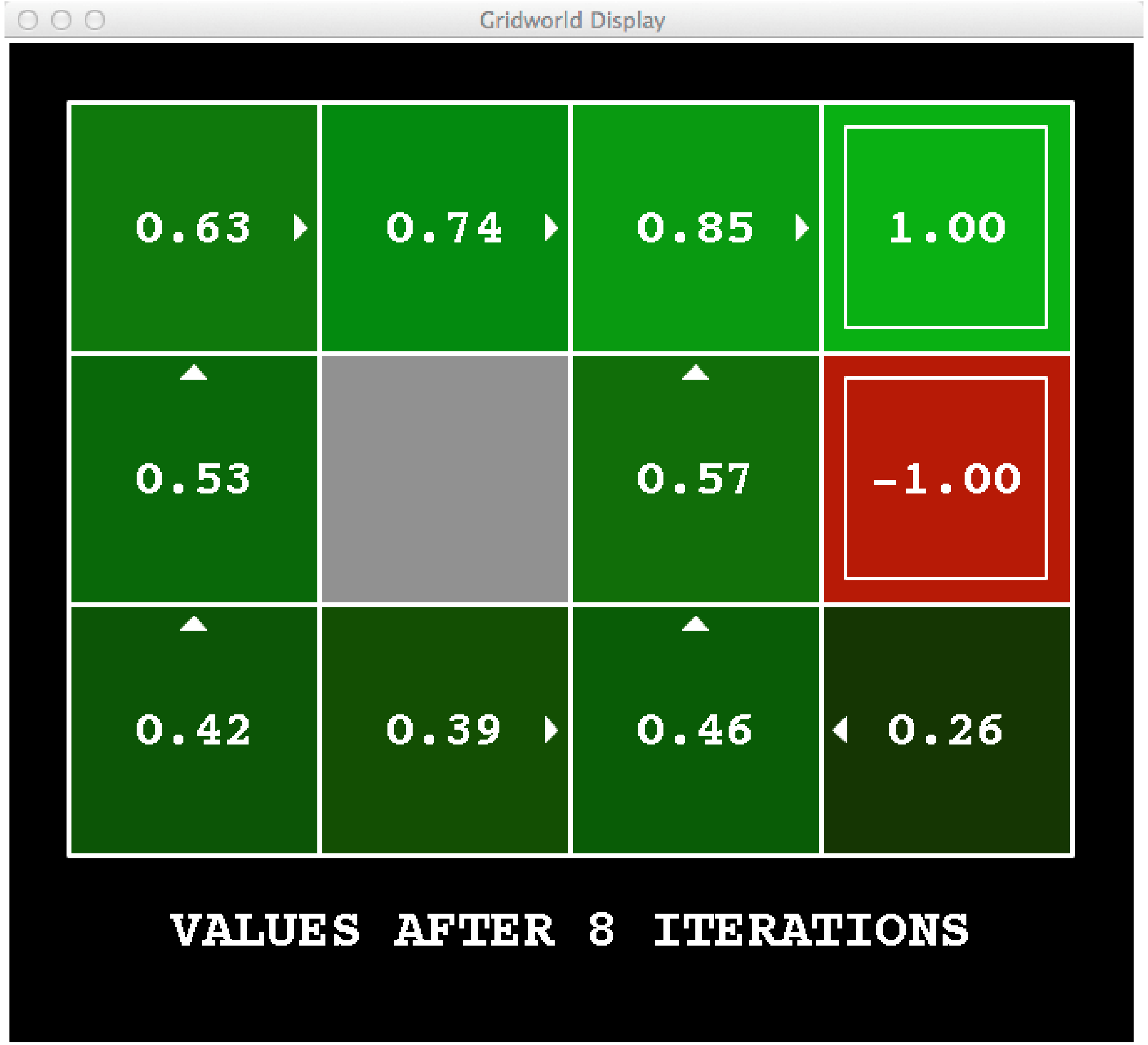
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=7



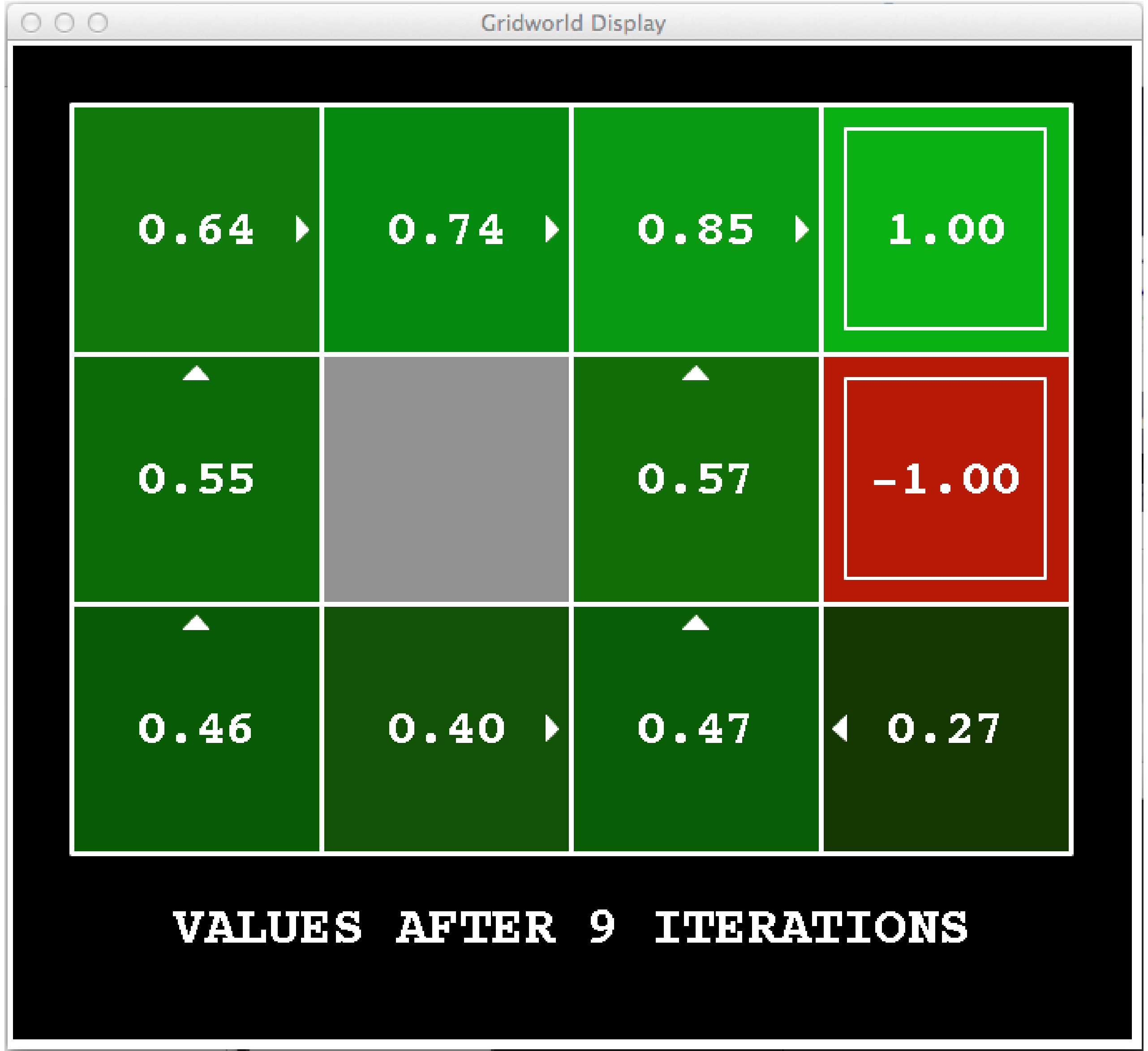
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=8



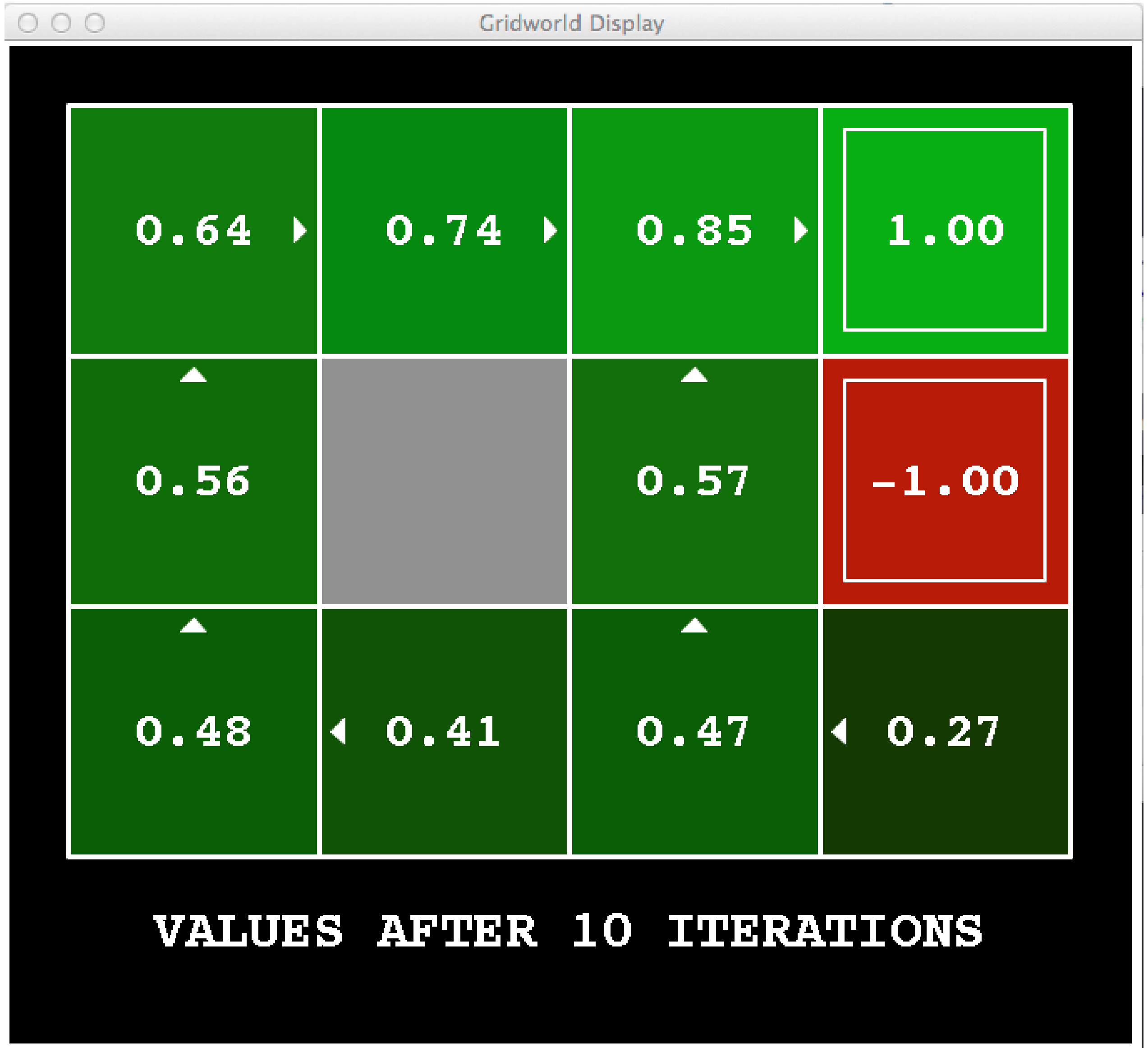
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=9



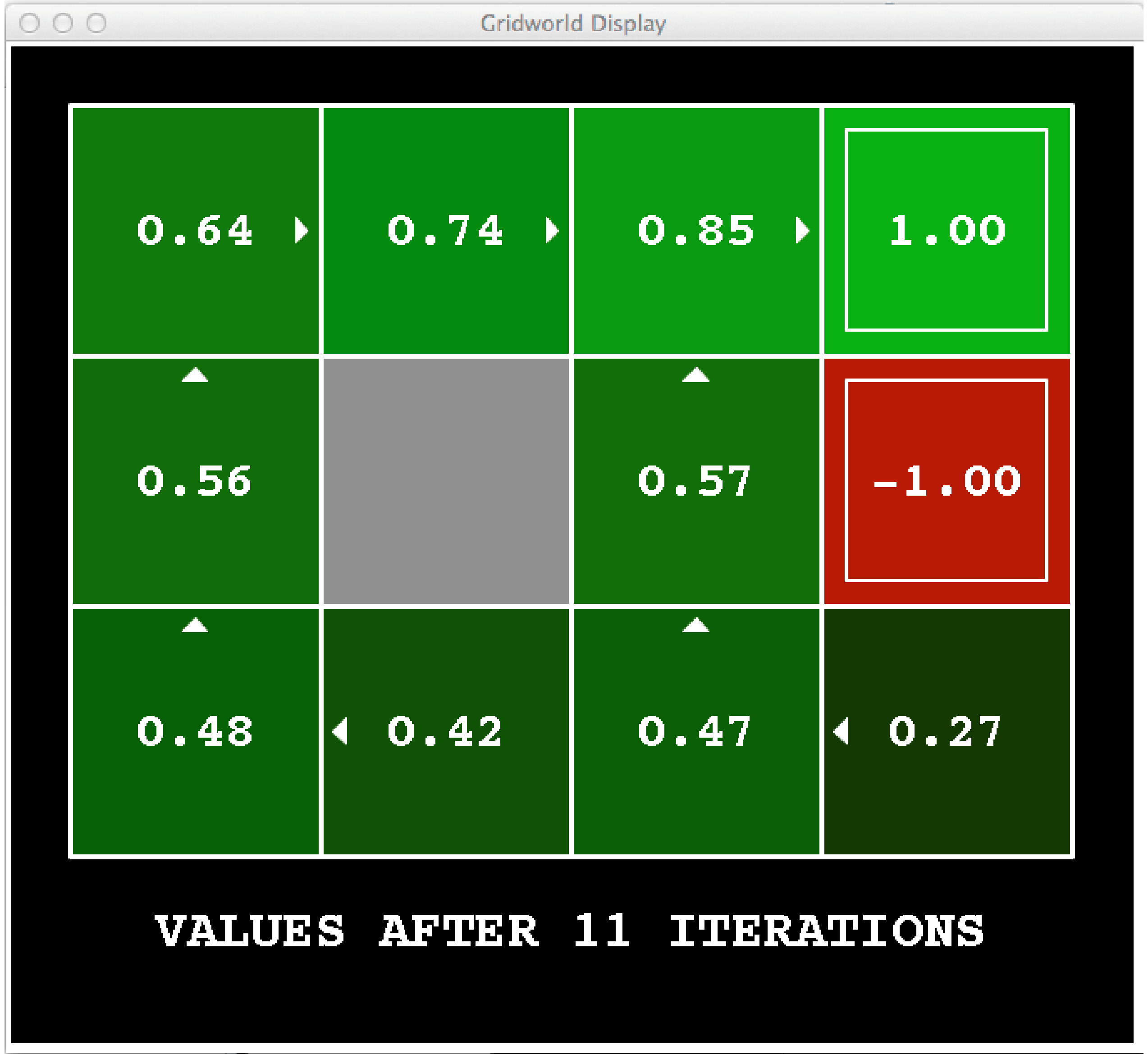
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=10



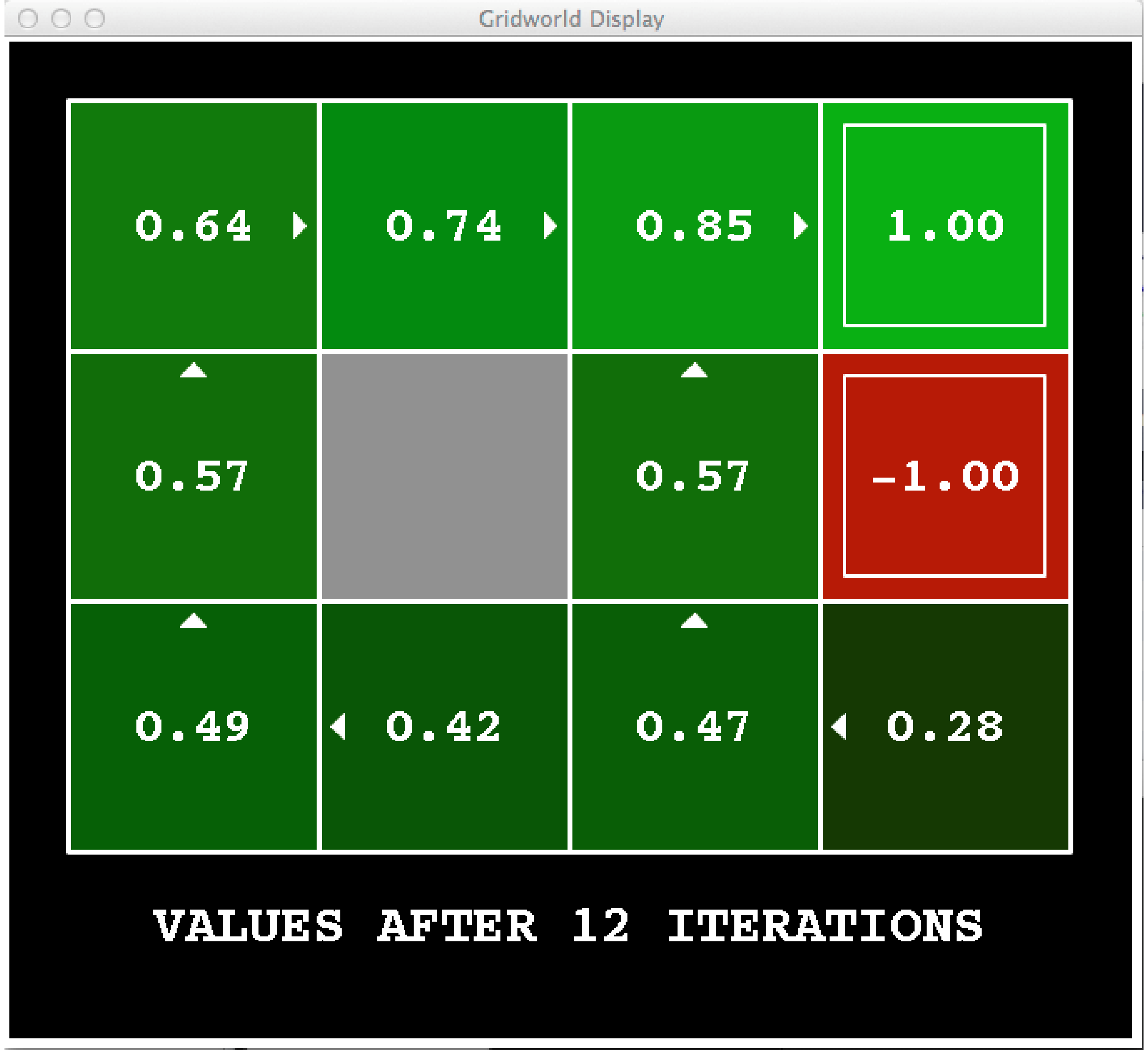
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=11



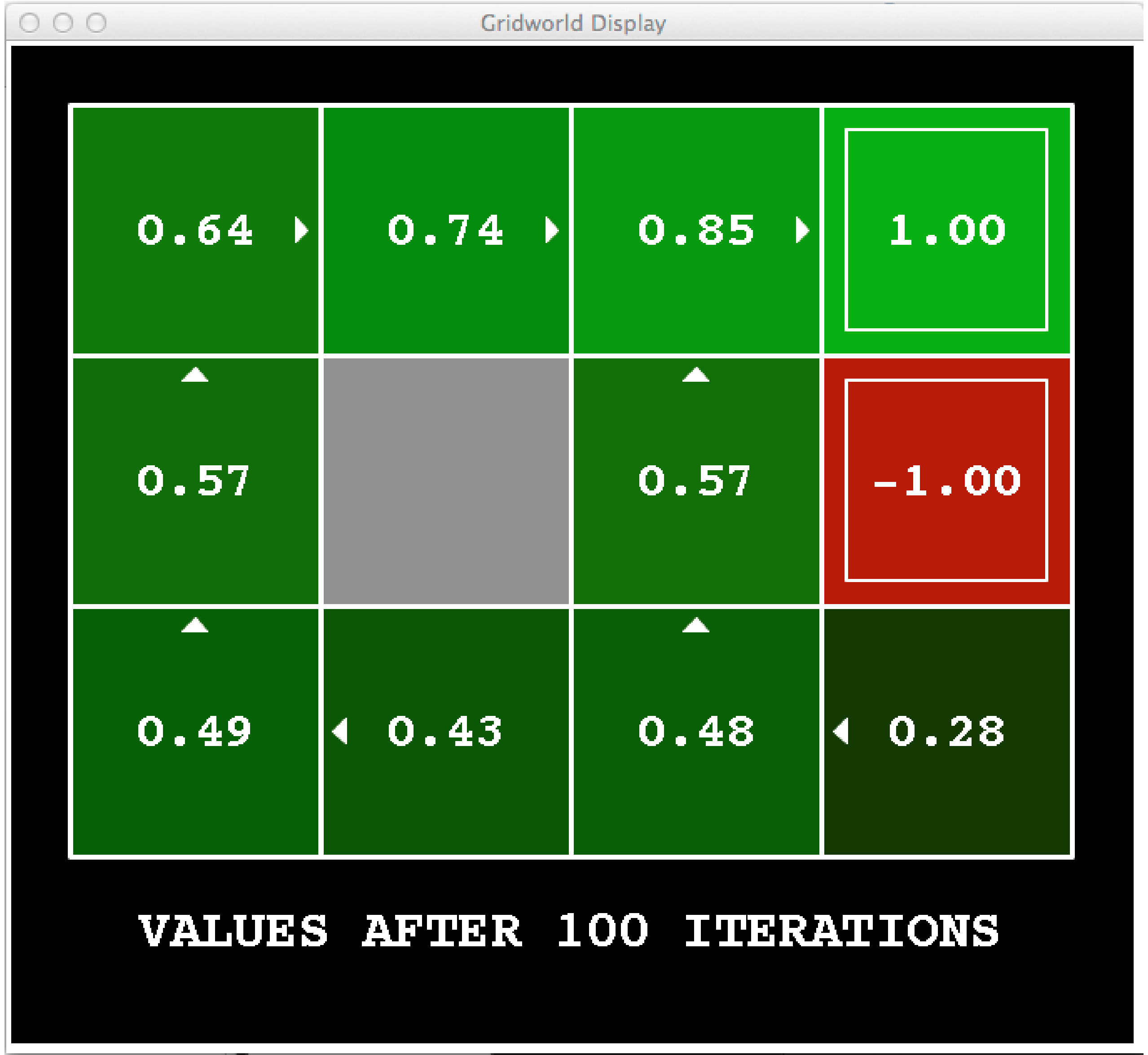
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=12



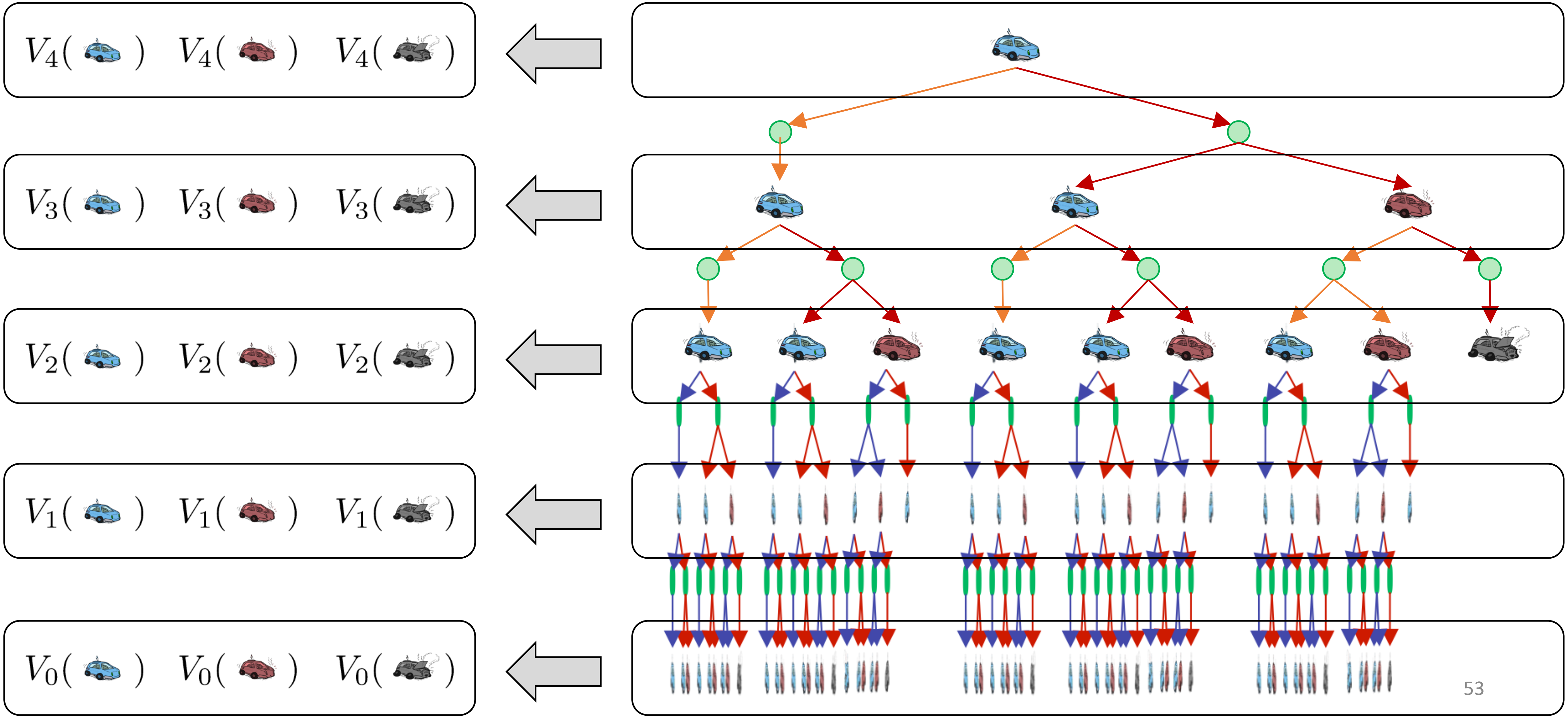
Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=100

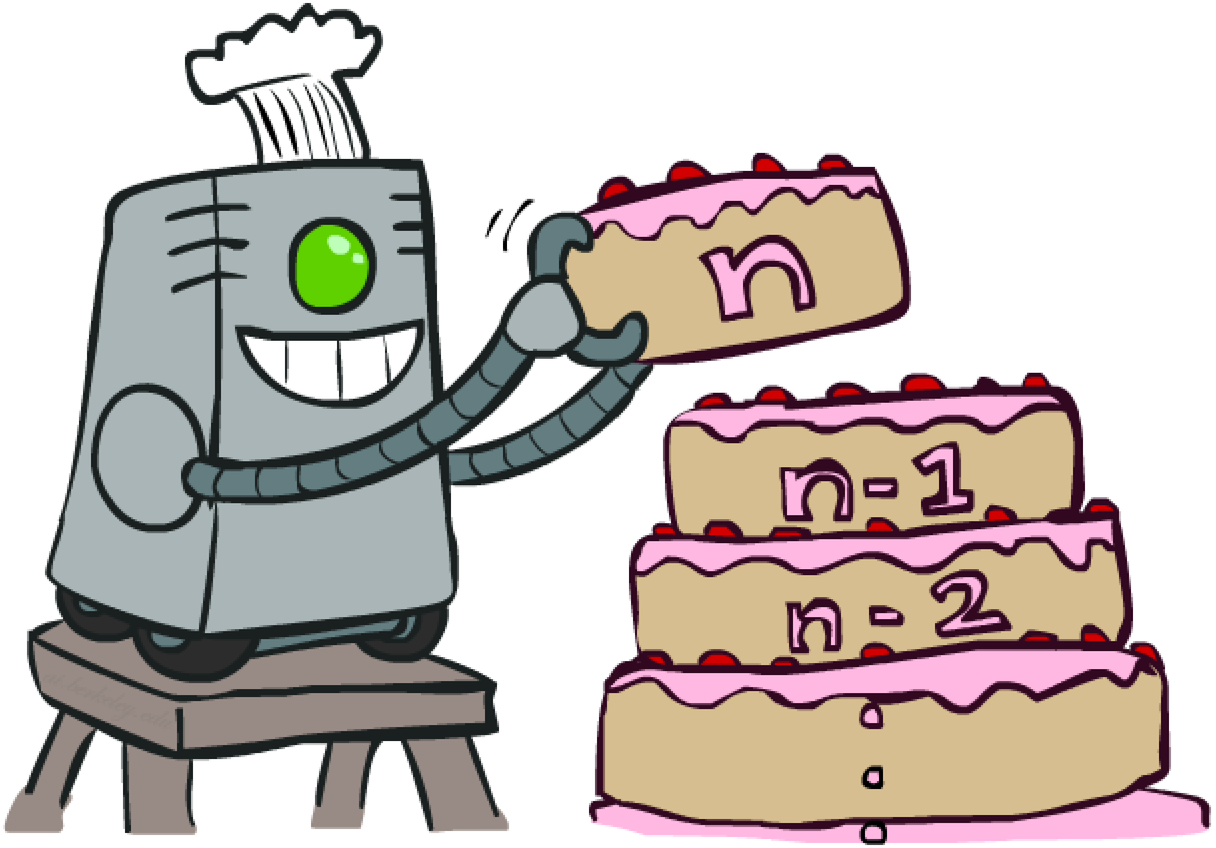


Noise = 0.2
Discount = 0.9
Living reward = 0

Time-Limited Values: Computing



Value Iteration

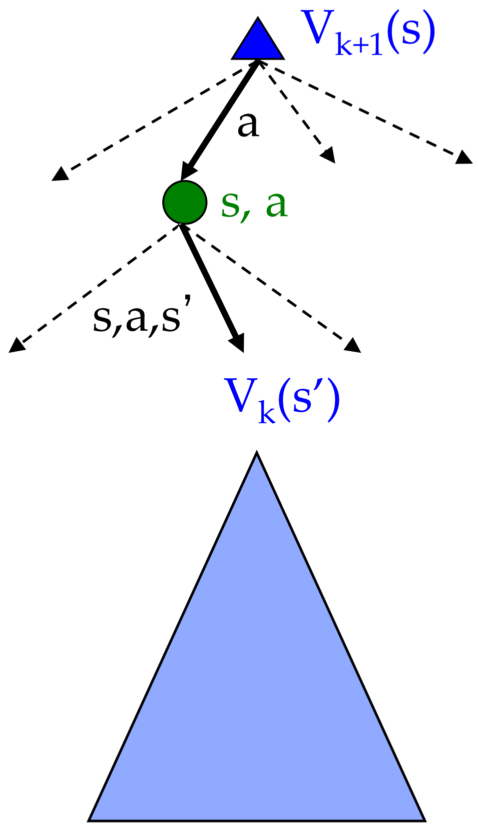


Value Iteration

- 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:

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

- 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

V_2

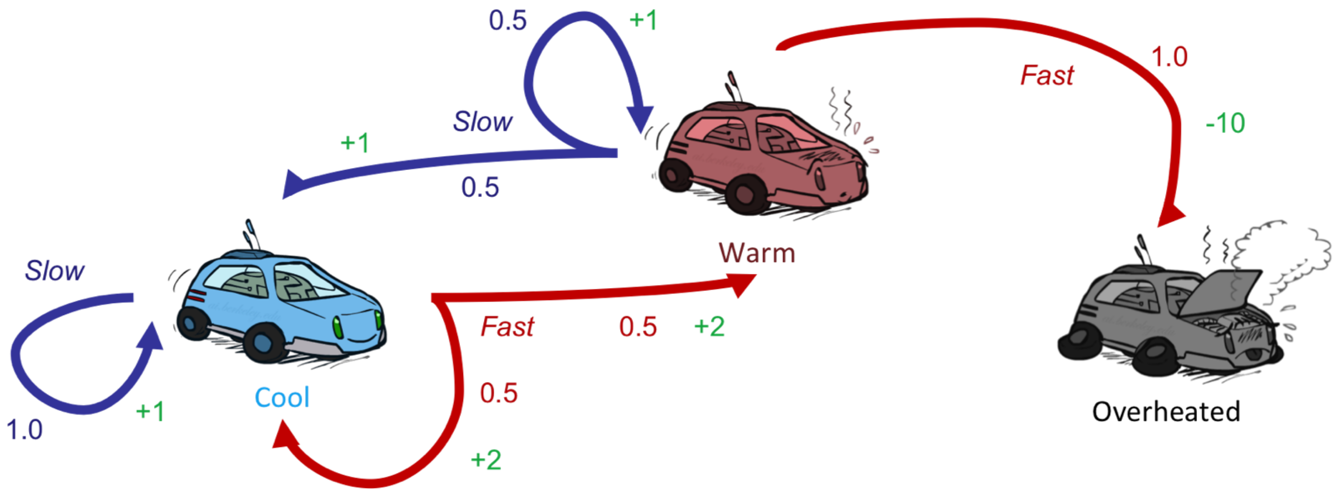
V_1

S: 1

F: .5*2+.5*2=2

V_0

0 0 0

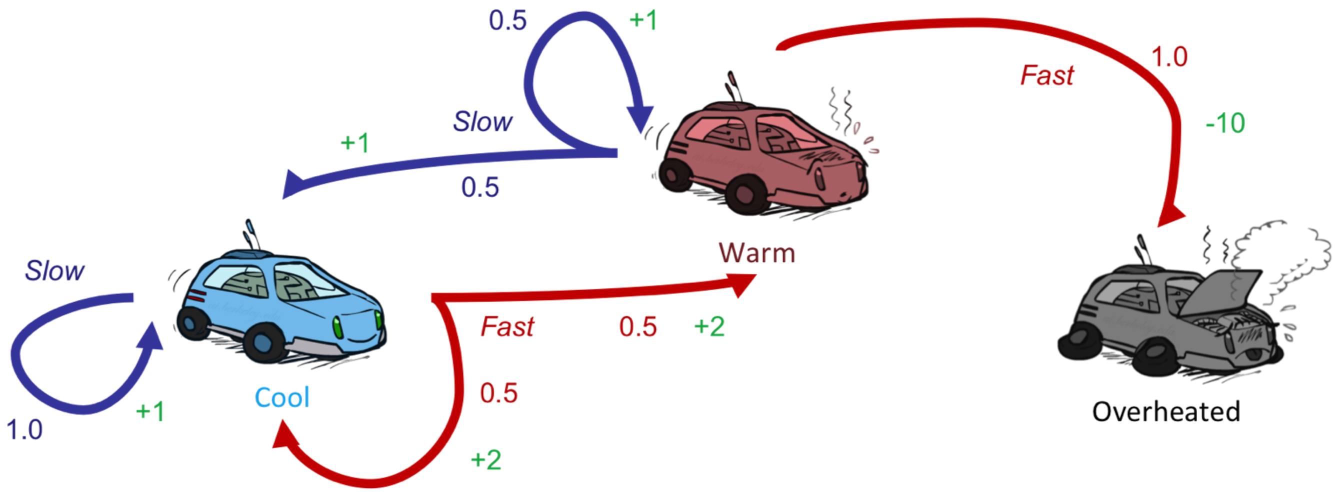


Assume no discount!

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

Example 2

V_2			
V_1	2	S: .5*1+.5*1=1 F: -10	
V_0	0	0	0



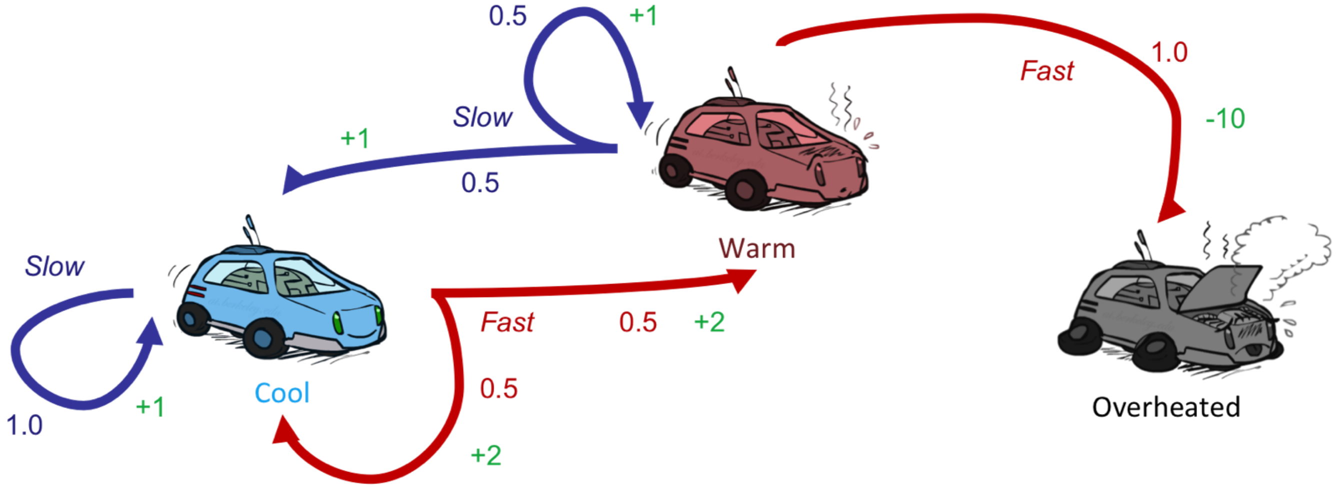
Assume no discount!

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

Example 3



V_2			
V_1	2	1	0
V_0	0	0	0

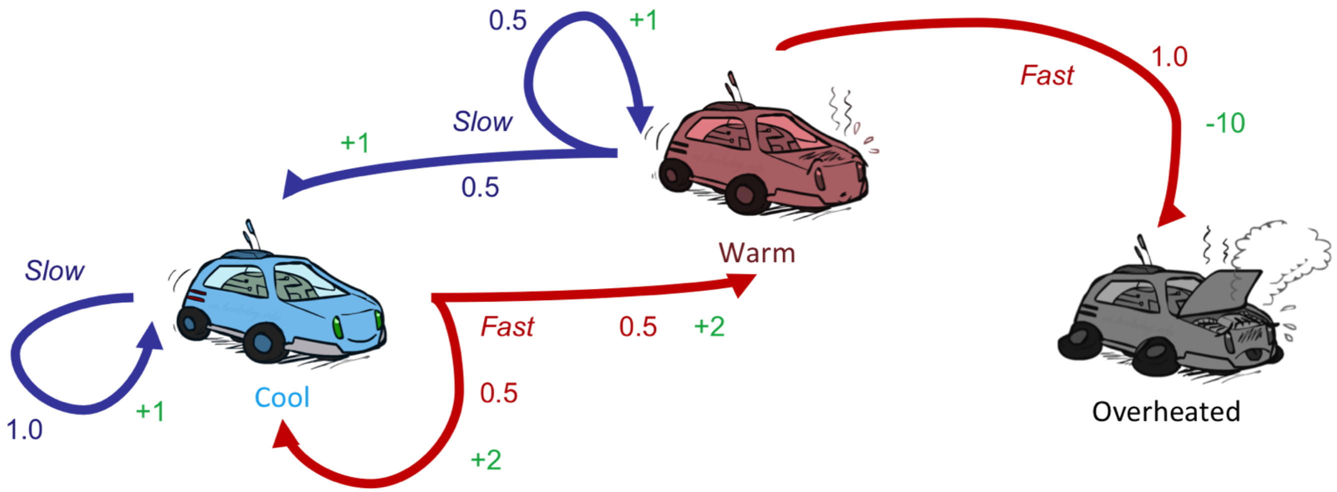


Assume no discount!

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

Example 4

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

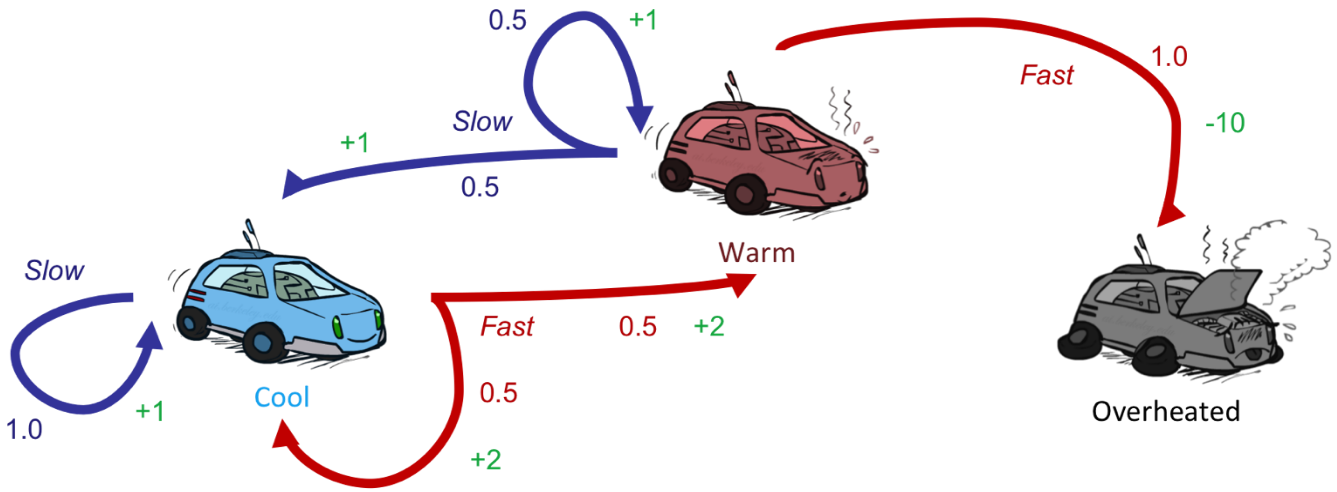


Assume no discount!

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

Example 5

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

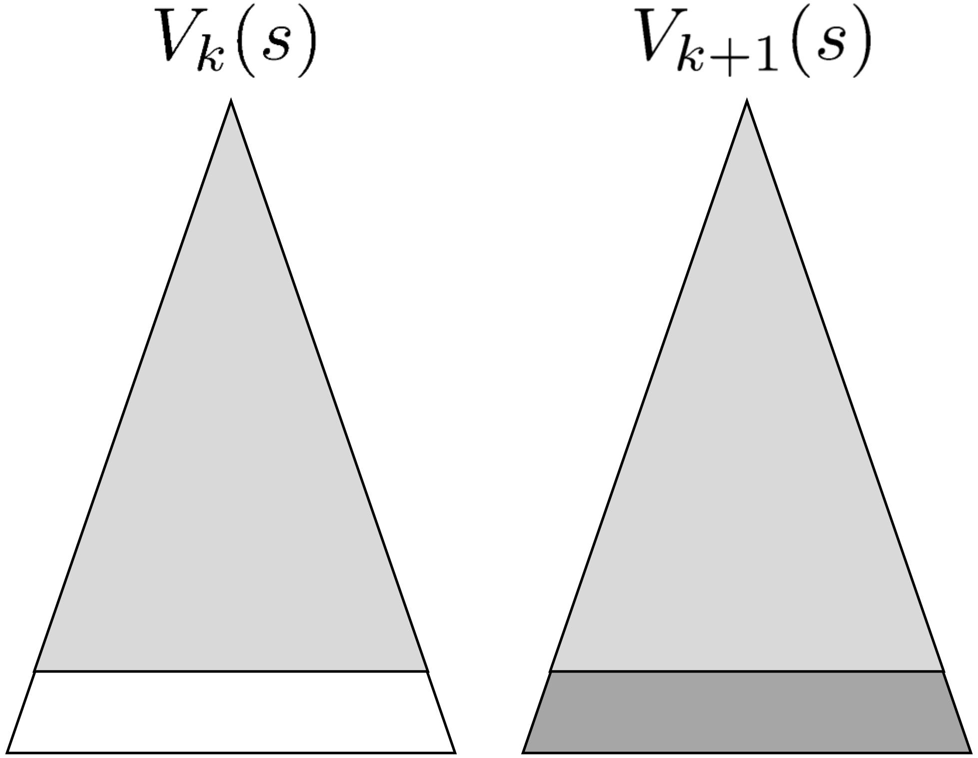


Assume no discount!

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

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
- Proof 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 γ^k max|R| different
  - So as k increases, the values converge

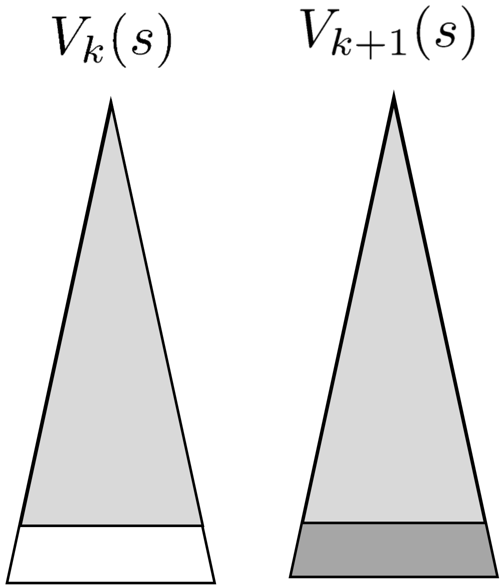


Convergence 2

- $V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$
  $V'_{k+1}(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V'_k(s')]$
- If $|V_k(s) - V'_k(s)| \le \epsilon$, then
  $$|V_{k+1}(s) - V'_{k+1}(s)| \le \gamma\epsilon$$
- Then treat $V_{k+1}(s) =: V'_k(s)$
- Note $\left|V_1(s) = \max_a \sum_{s'} T(s,a,s') R(s,a,s')\right| \le R_{\max}$

  that is $|V_1(s) - V_0(s)| \le R_{\max}$, then $|V_2(s) - V_1(s)| \le \gamma R_{\max}$ and

  $$|V_{k+1}(s) - V_k(s)| \le \gamma^k R_{\max}$$



Value Iteration (Revisited)

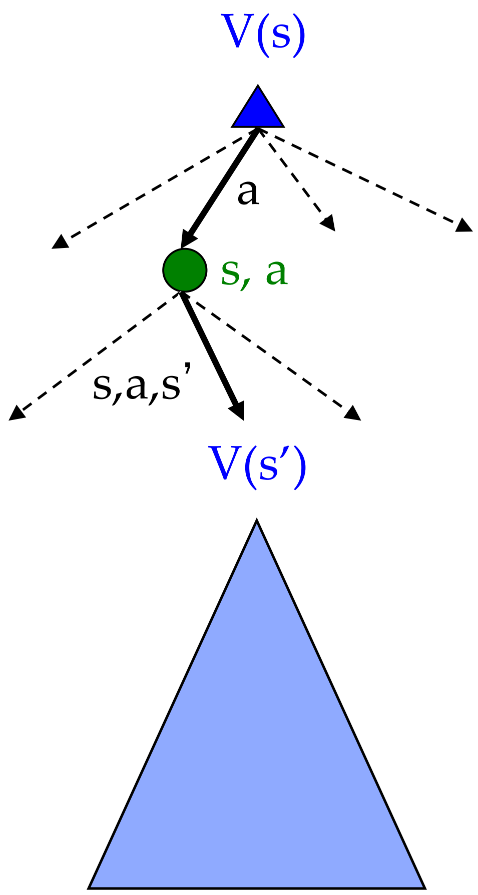
- Bellman equations characterize the optimal values:

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

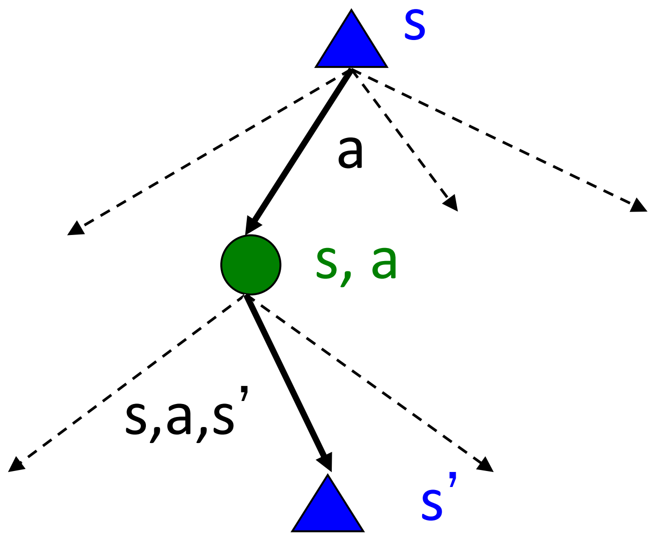
- Value iteration computes them:

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

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



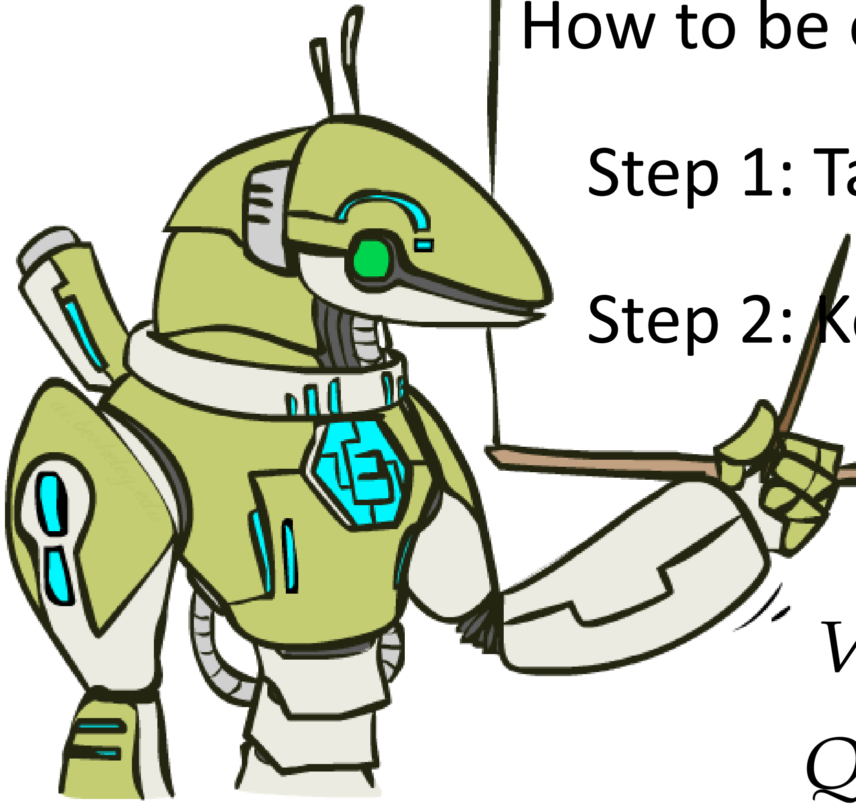
Value Iteration - Implementation



- Init:
  - $\forall s: \quad V(s) = 0$
- Iterate:
  - $\forall s: \quad V_{new}(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
  - $V = V_{new}$

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



How to be optimal:

Step 1: Take correct first action

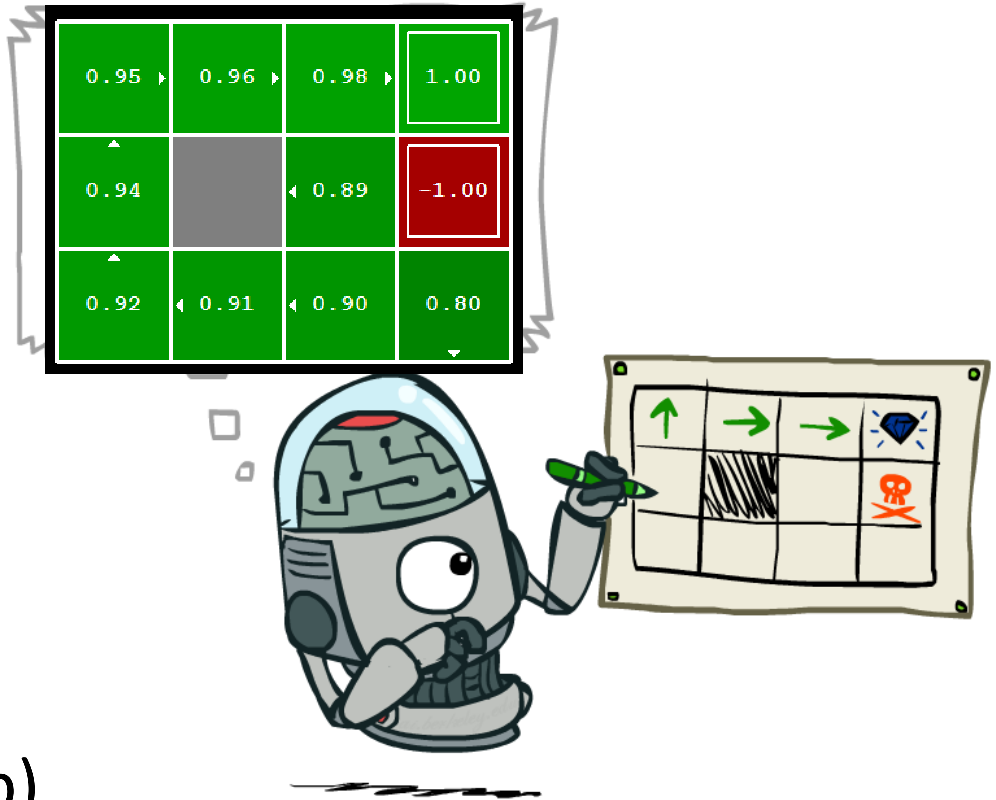
Step 2: Keep being optimal

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

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

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

Policy Extraction: Computing Actions from Values



- Let's imagine we have the optimal values V*(s)

- How should we act?
  - It's not obvious!

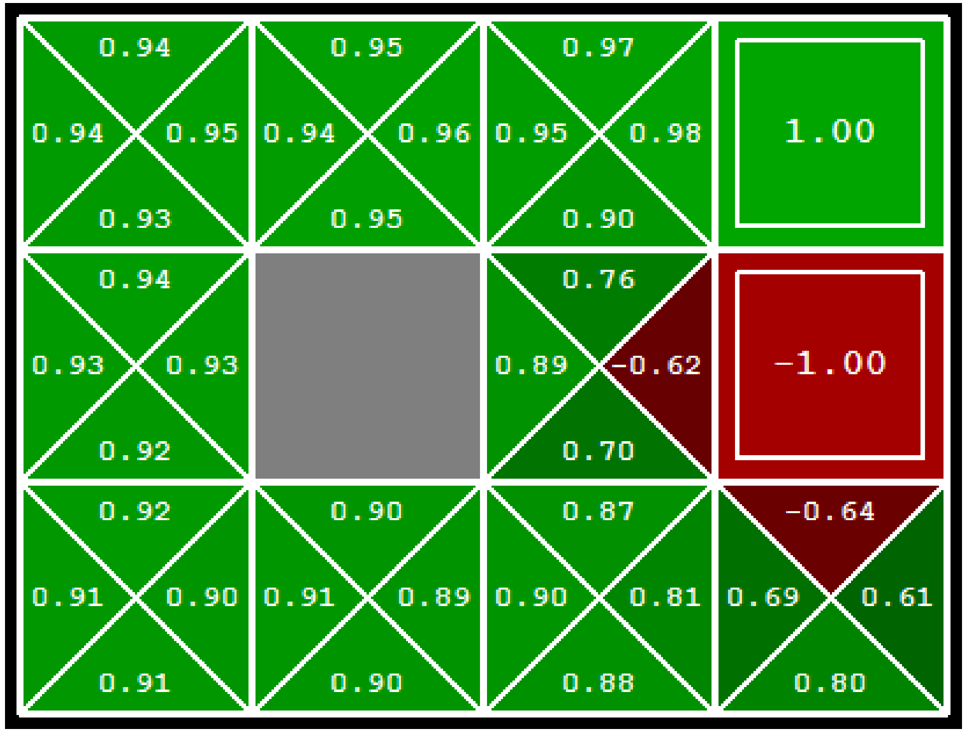
- We need to do a mini-expectimax (one step)

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

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

Policy Extraction: Computing Actions from Q-Values

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

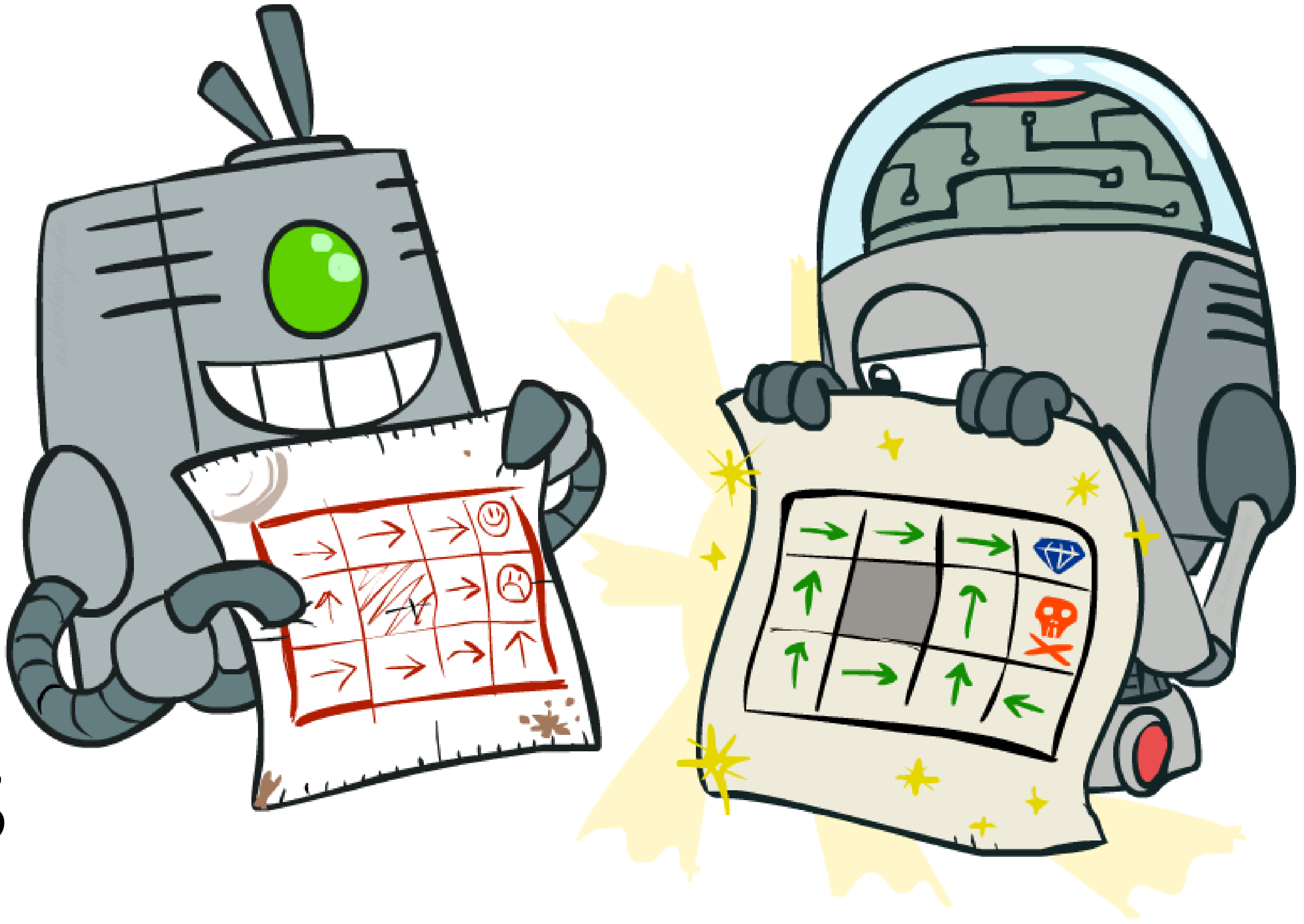


- How should we act?
  - Completely trivial to decide!

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

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

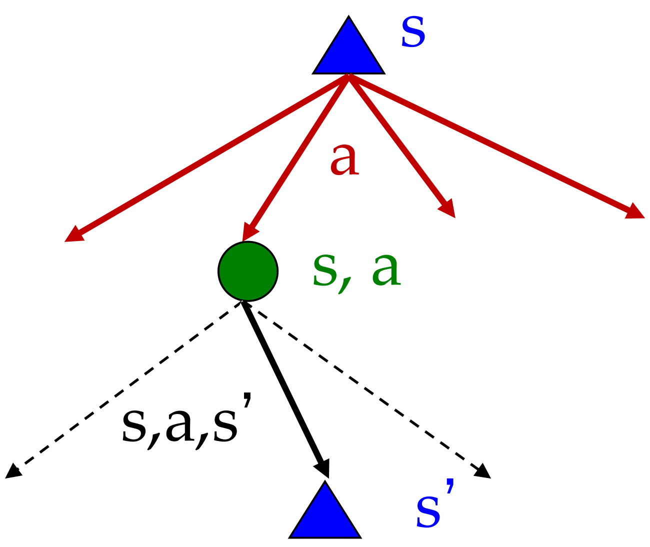
Policy Methods



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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

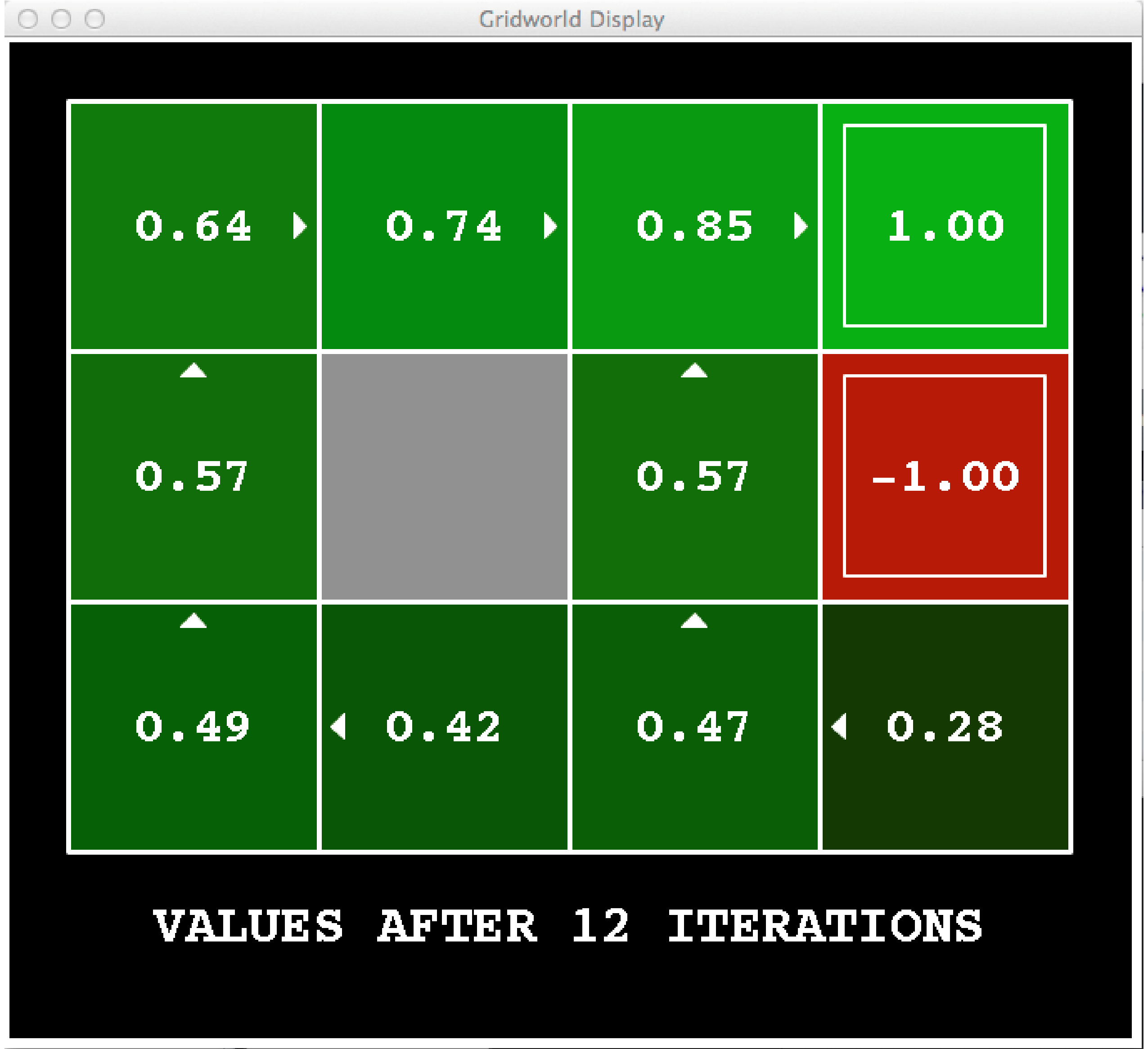


- Problem 1: It's slow – $O(S^2A)$ per iteration

- Problem 2: The "max" at each state rarely changes

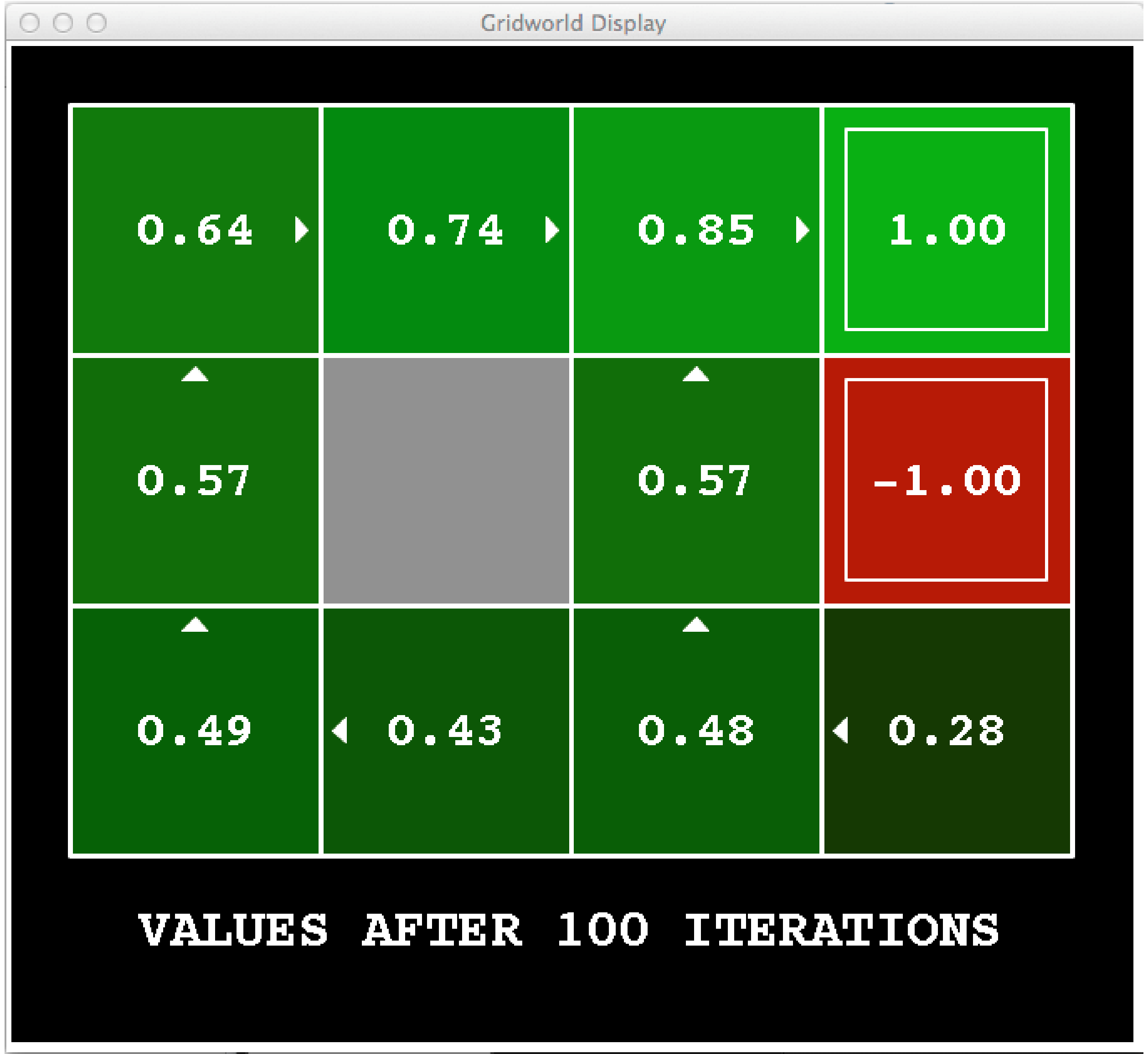
- Problem 3: The policy often converges long before the values

Gridworld: k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=100



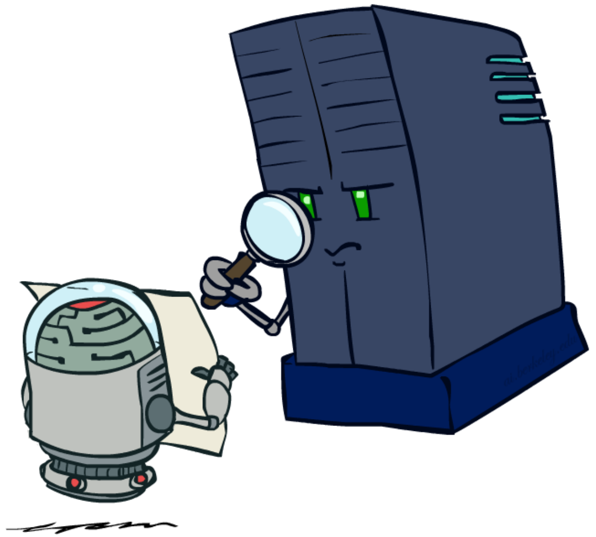
Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

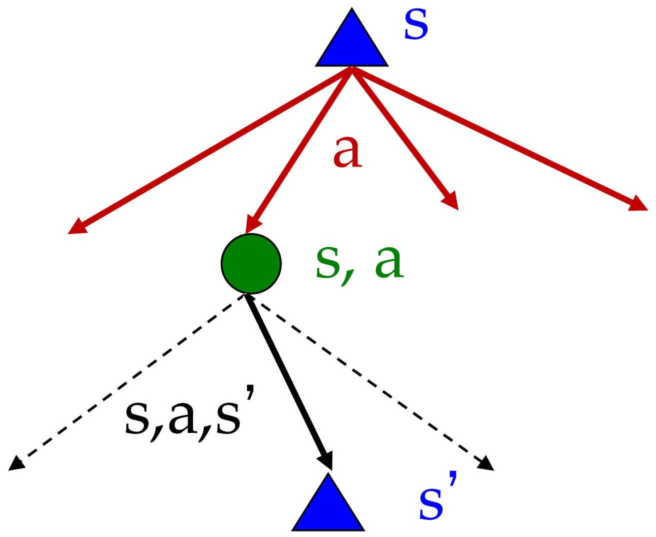
- Alternative approach for optimal values:
  - Step 1: Policy Evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy Improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is Policy Iteration
  - It’s still optimal!
  - Can converge (much) faster under some conditions

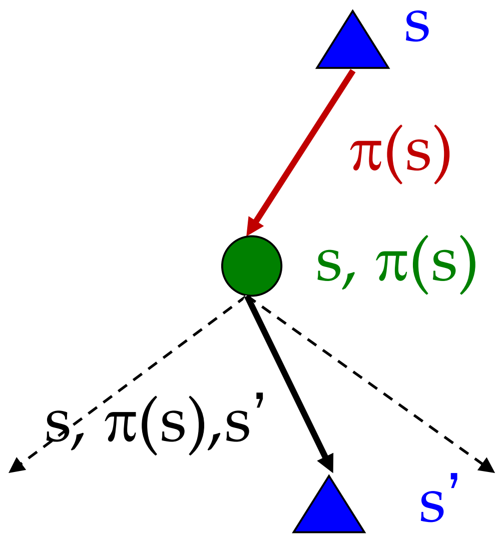
Policy Evaluation: Fixed Policies



Do the optimal action



Do what π says to do



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

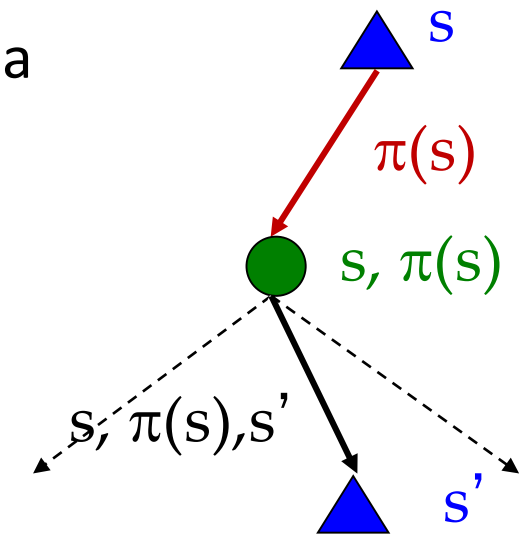
Policy Evaluation: Utilities for a Fixed Policy

- 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:

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

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



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

Policy Evaluation: Implementation

- How do we calculate the V's for a fixed policy π?

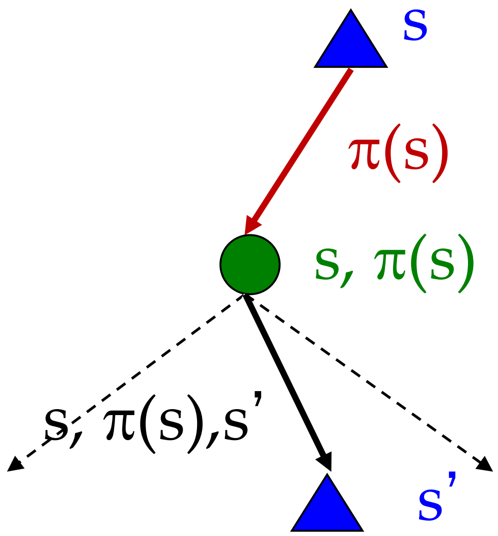
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

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

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

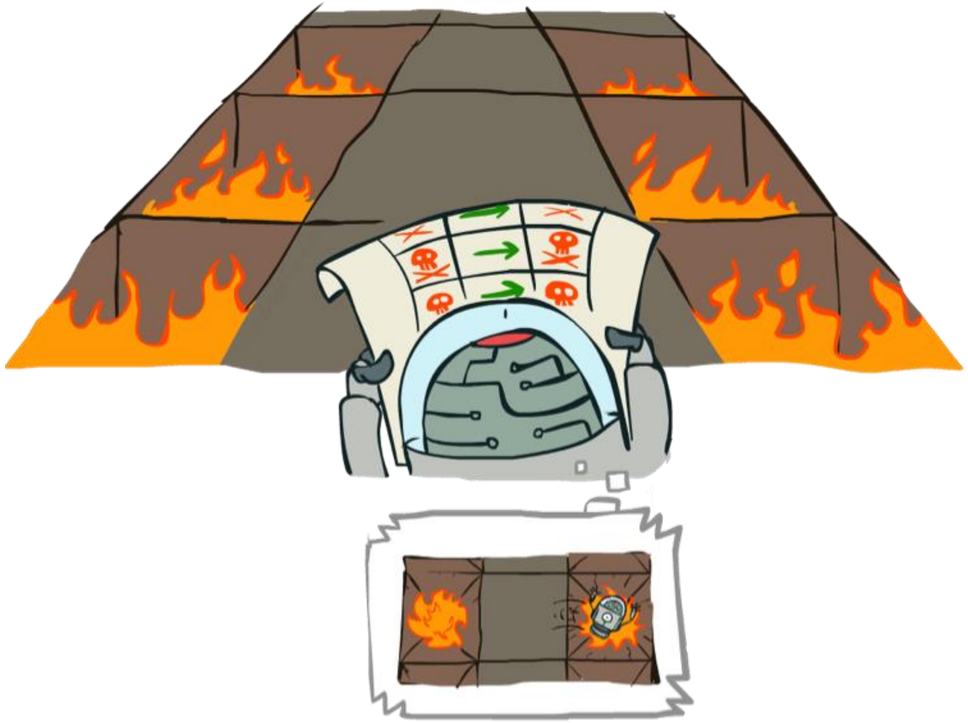
- Efficiency: $O(S^2)$ per iteration

- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with MATLAB (or your favorite linear system solver)

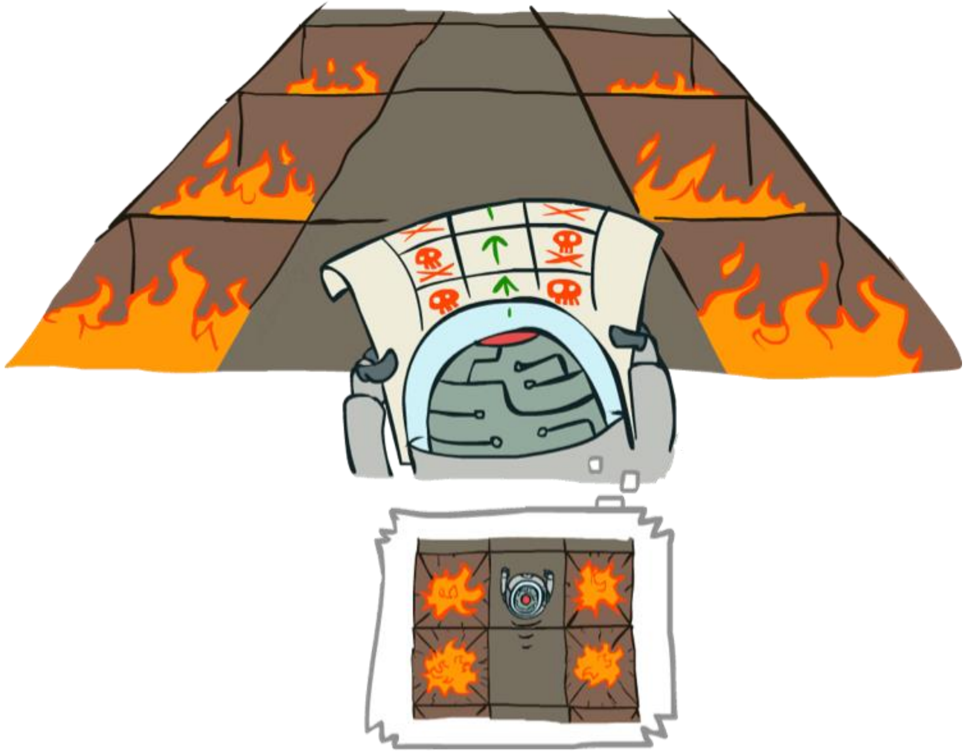


Example: Policy Evaluation

Always Go Right



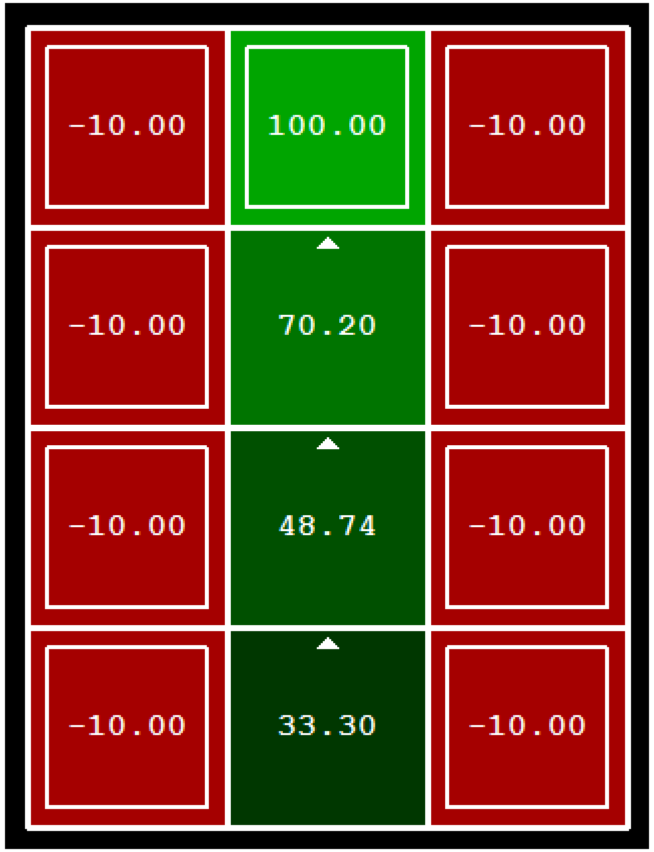
Always Go Forward



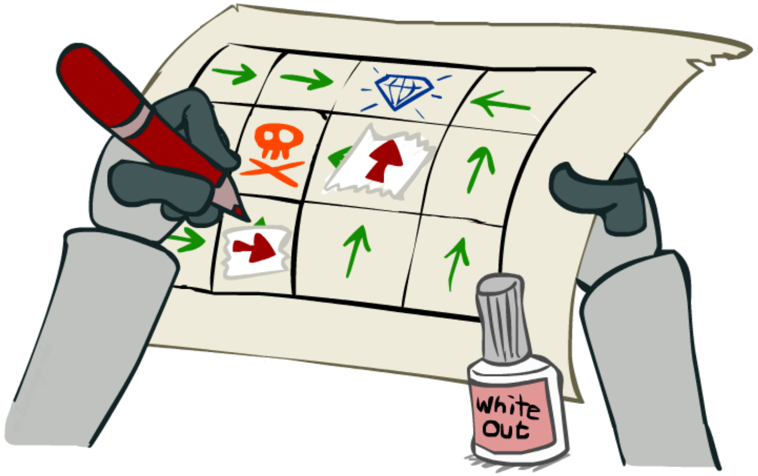
Example: Policy Evaluation 2

Always Go Right

Always Go Forward



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 (why? exercise) policy using policy extraction
  - One-step look-ahead:

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

Summary of Two Methods for Solving MDPs

- Value iteration + policy extraction
  - Step 1: Value iteration: calculate values for all states by running one ply of the Bellman equations using values from previous iteration **until convergence**
  - Step 2: Policy extraction: compute policy by running one ply of the Bellman equations using values from value iteration
- Policy iteration
  - Step 1: Policy evaluation: calculate values for some fixed policy (not optimal values!) **until convergence**
  - Step 2: Policy improvement: update policy by running one ply of the Bellman equations using values from policy evaluation
  - Repeat steps until policy converges

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)
  - The new policy will be better (or we’re done)

- Both are dynamic programs for solving MDPs

Summary

Shuai Li

https://shuaili8.github.io

- Markov Decision Processes
  - Probabilistic transition, Markov, policies
  - Utilities of sequences
  - Optimal quantities, value of states, Bellman equations
- Value iteration
  - Time-limited values, convergence
  - Policy extraction
- Policy iteration
  - Policy evaluation
  - Policy improvement

Questions?