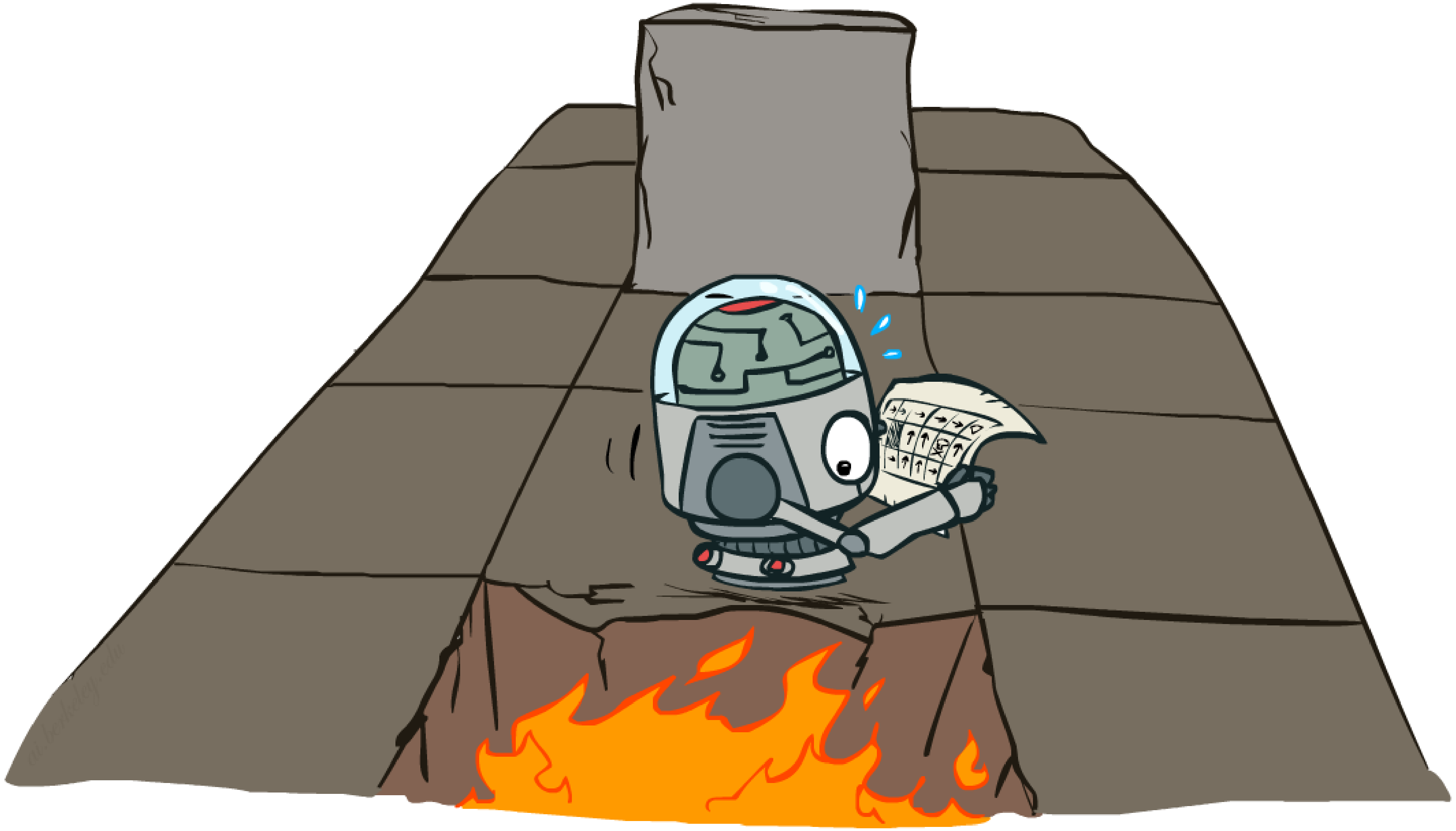
Announcements

- Homework 4: MDPs (today's topic)
  - Due Monday 9/26 at 11:59pm.
- Project 2: Multi-Agent Pacman
  - Has been released, due Friday 9/30 at 5:00pm.
- Contest 2, due at a time TBA, but soon after project 2.
  - Small tweak from contest 1.
- Survey on how we're doing so far has been released.
  - Due Saturday 9/24 at 11:59 PM.
  - +1 project point added to total (equivalent to one contest).
- Midterm: Oct 6
  - **Time to start studying.**
  - Recommended approach:
    - Work through problems alone.
    - Get together in group of 3-6, and whiteboard attempted solutions.
    - Interrupt each other!
  - Fill out midterm conflict form ASAP. Makeup at 8 AM.

CS 188: Artificial Intelligence

Markov Decision Processes II

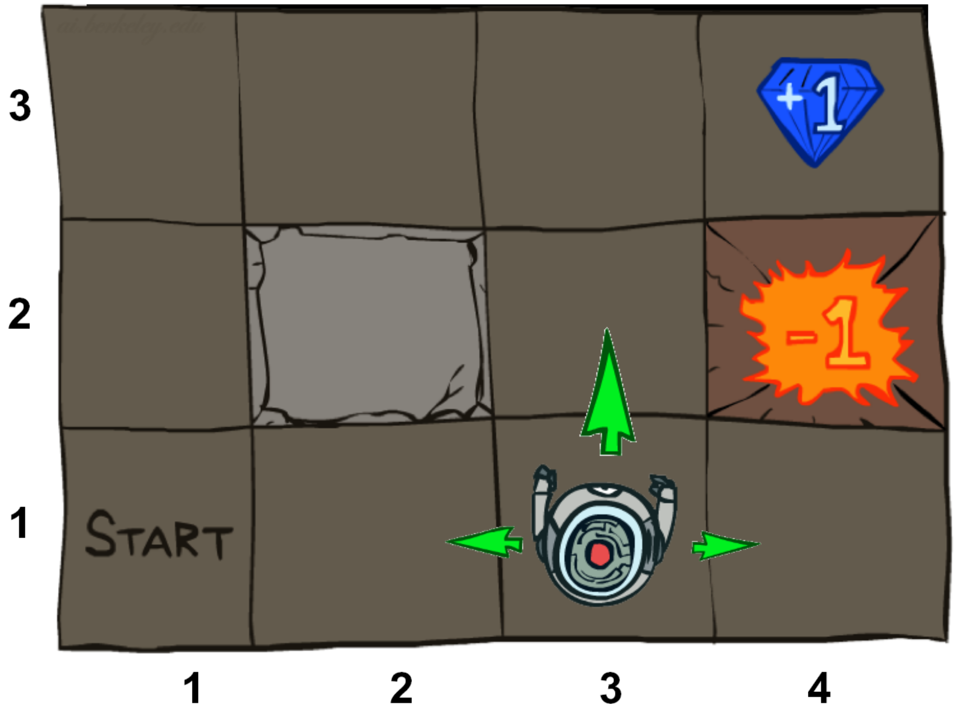


Instructors: Dan Klein and Pieter Abbeel --- University of California, Berkeley

[These slides were created by Dan Klein, Pieter Abbeel, Josh Hug for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

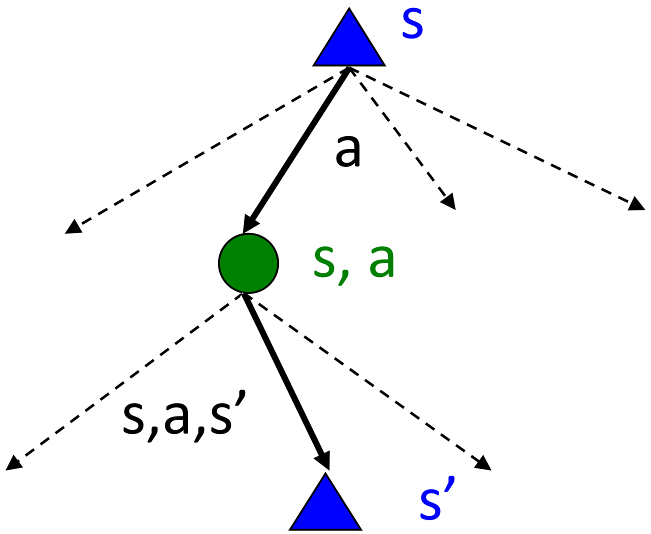
Example: Grid World

- 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% chance: agent goes the way it wants to go (e.g. the action North takes the agent North)
  - 10% chance: agent steps left (e.g. North, but goes West)
  - 10% chance: agent steps right (e.g. North, but goes East)
  - If there is a wall in the direction the agent would have been taken, the agent stays put
  - 0% chance to go backwards
- 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



Recap: MDPs

- Markov decision processes:
  - States S
  - Actions A
  - Transitions P(s’|s,a) (or T(s,a,s’))
  - Rewards R(s,a,s’) (and discount γ)
  - Start state s_0



- Quantities:
  - Policy = map of states to actions
  - Utility = sum of discounted rewards
  - Values = expected future utility from a state (max node)
  - Q-Values = expected future utility from a q-state (chance node)

Optimal Quantities

- The value (utility) of a state s:

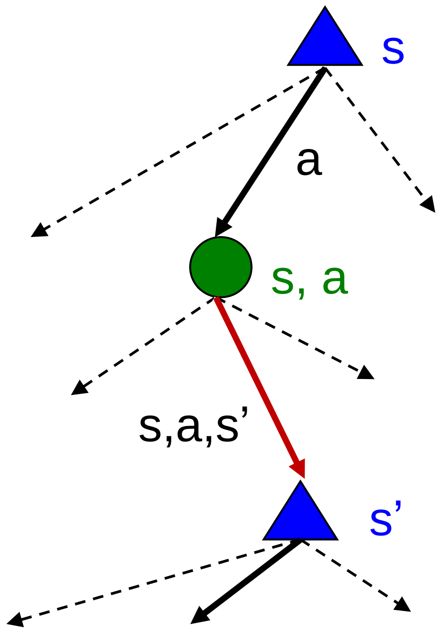
  $V^*(s)$ = expected utility starting in s and acting optimally

- 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



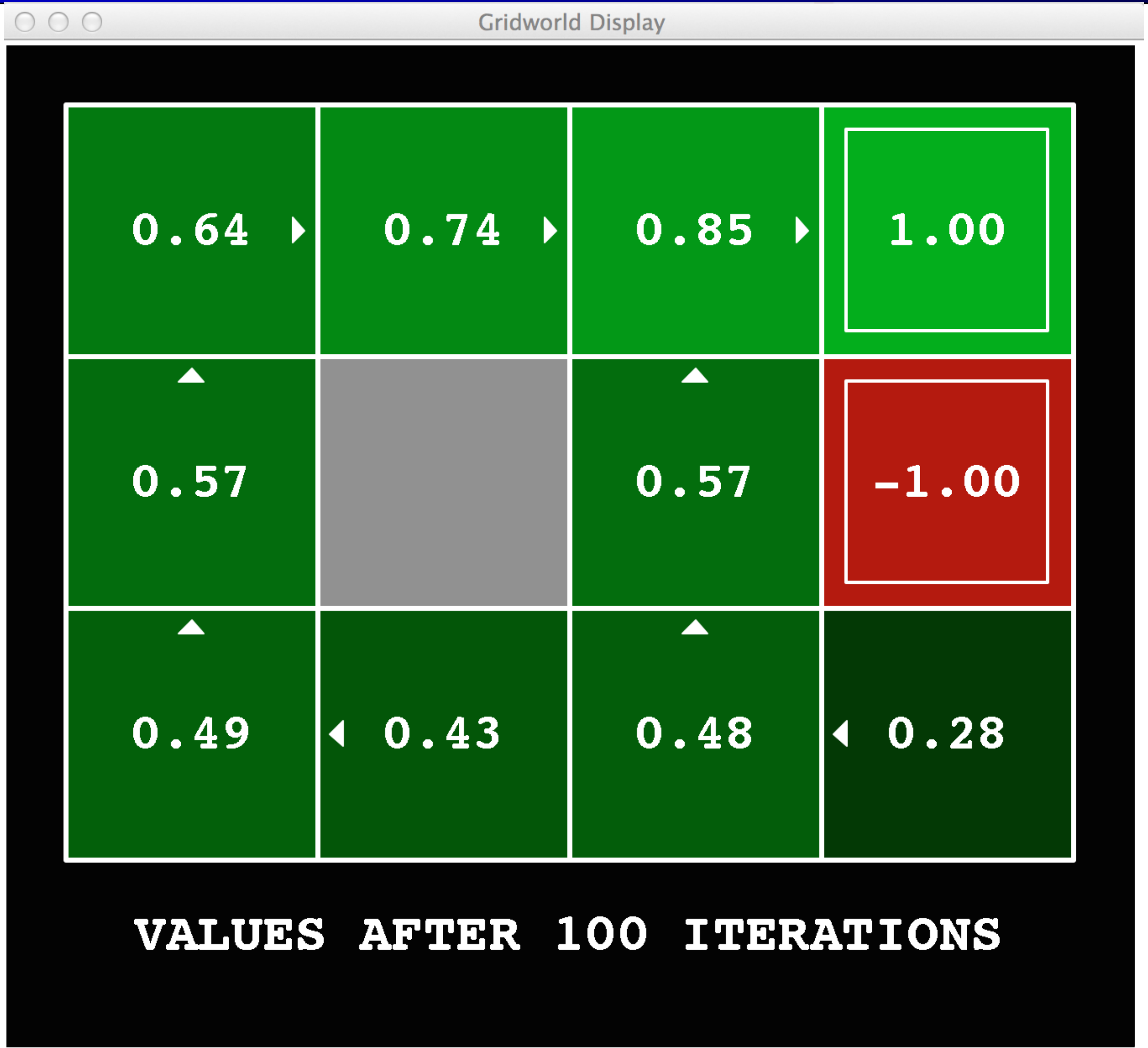
s is a *state*

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

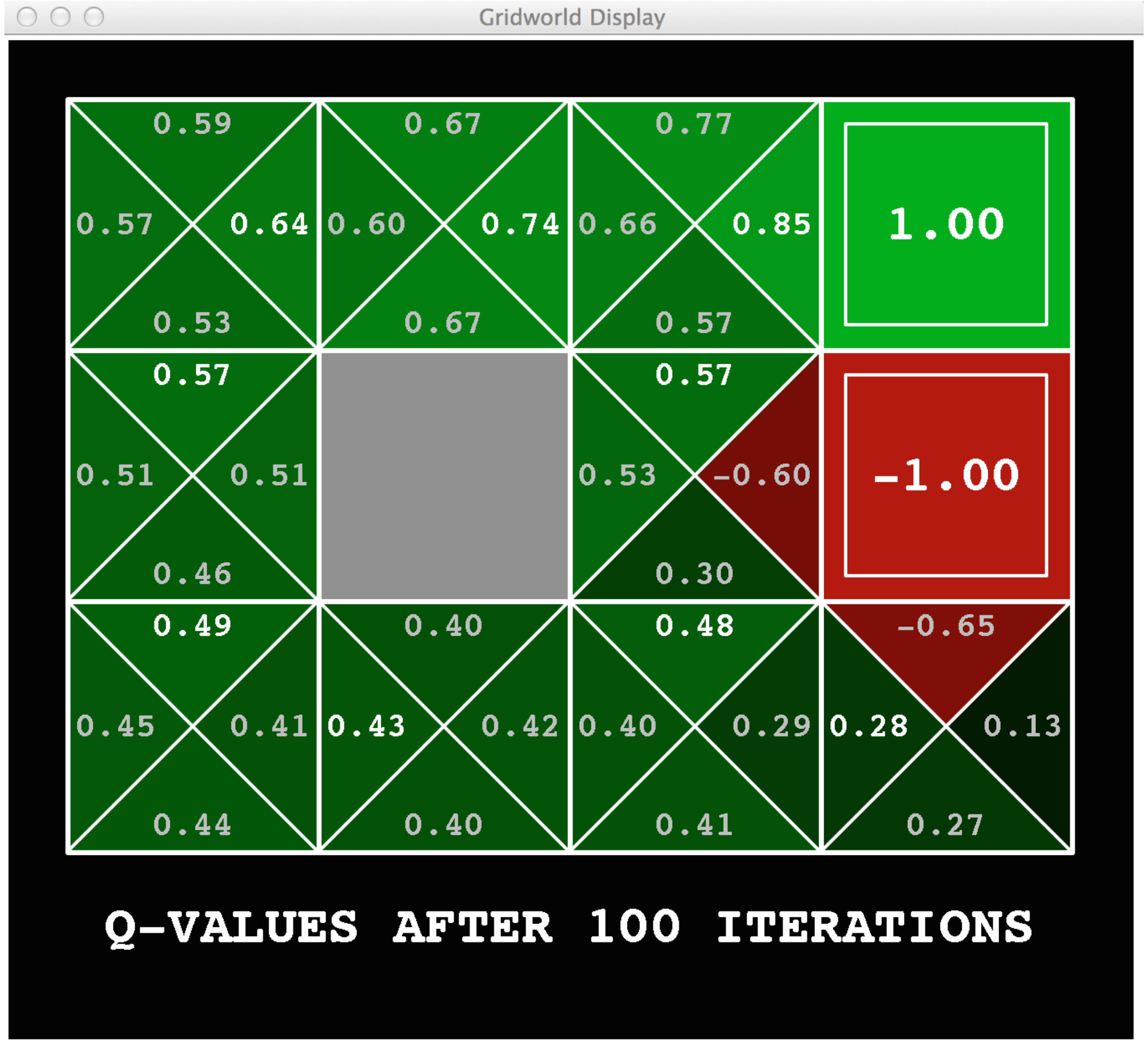
(s,a,s') is a *transition*

[Demo: gridworld values (L9D1)]

Gridworld Values V*

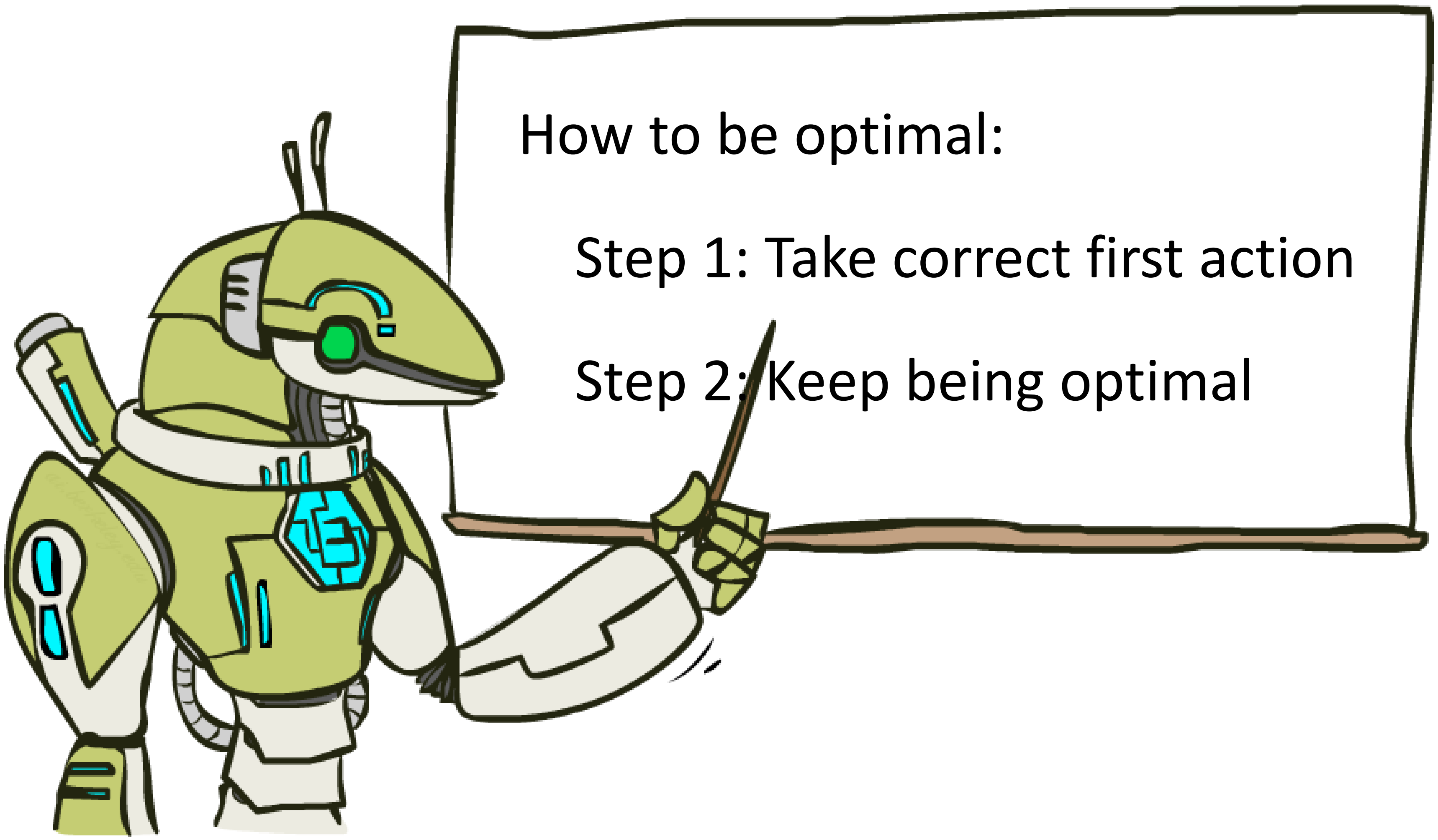


Gridworld: Q*



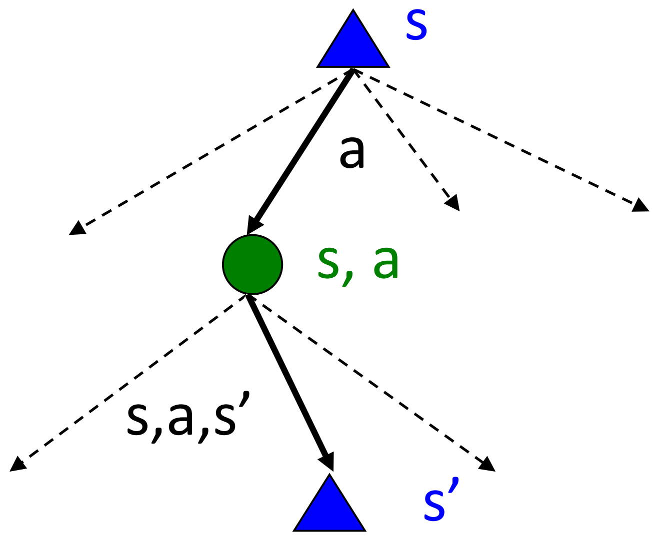
Gridworld Display
0.59
0.57
0.64
0.53
0.67
0.60
0.74
0.67
0.77
0.66
0.85
0.57
1.00
0.57
0.51
0.51
0.46
0.57
0.53
-0.60
0.30
-1.00
0.49
0.45
0.41
0.44
0.40
0.43
0.42
0.40
0.48
0.40
0.29
0.41
-0.65
0.28
0.13
0.27
Q-VALUES AFTER 100 ITERATIONS

The Bellman Equations



The Bellman Equations

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



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

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

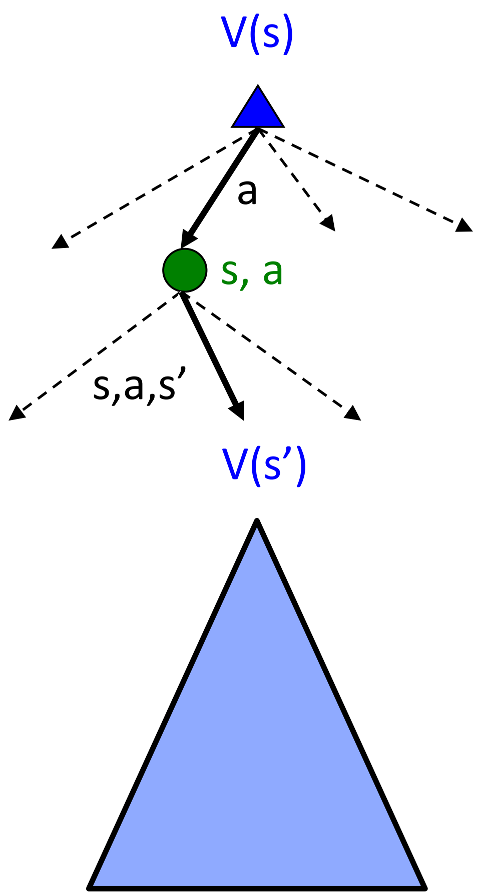
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

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

Value Iteration

- Bellman equations characterize the optimal values:

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



- Value iteration computes them:

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

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

Value Iteration (Alternate View)

- **Bellman Equation** gives us a way to characterize whether a vector of utilities is "correct", i.e the vector is equal to the expected utilities if we play optimally.

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

$V(s)$

2.375	1.375	0

If we plug in our V(cool), V(warm), and V(OH),and the equation is satisfied, then our vector V is the expected utility if we play optimally.

- Can use **value iteration** to iteratively transform an initially all zero vector into the "correct' vector. Or in other words, lets us compute the "correct" vector.

0	0	0

$V_0(s)$

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

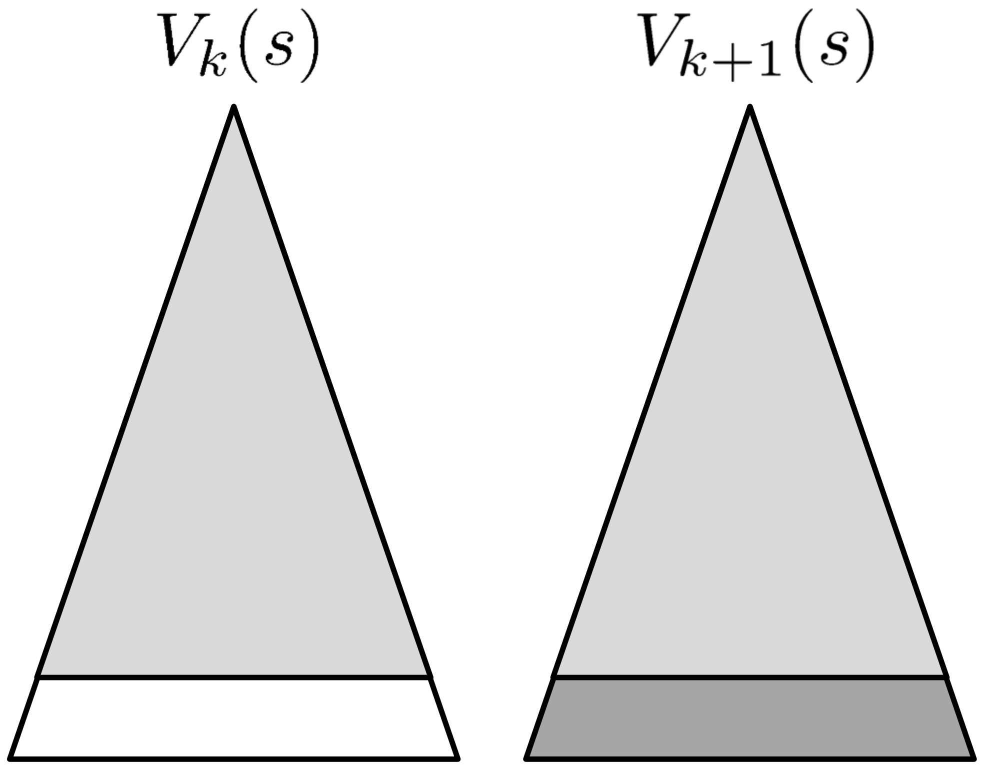
2	1	0

$V_1(s)$

…

Convergence

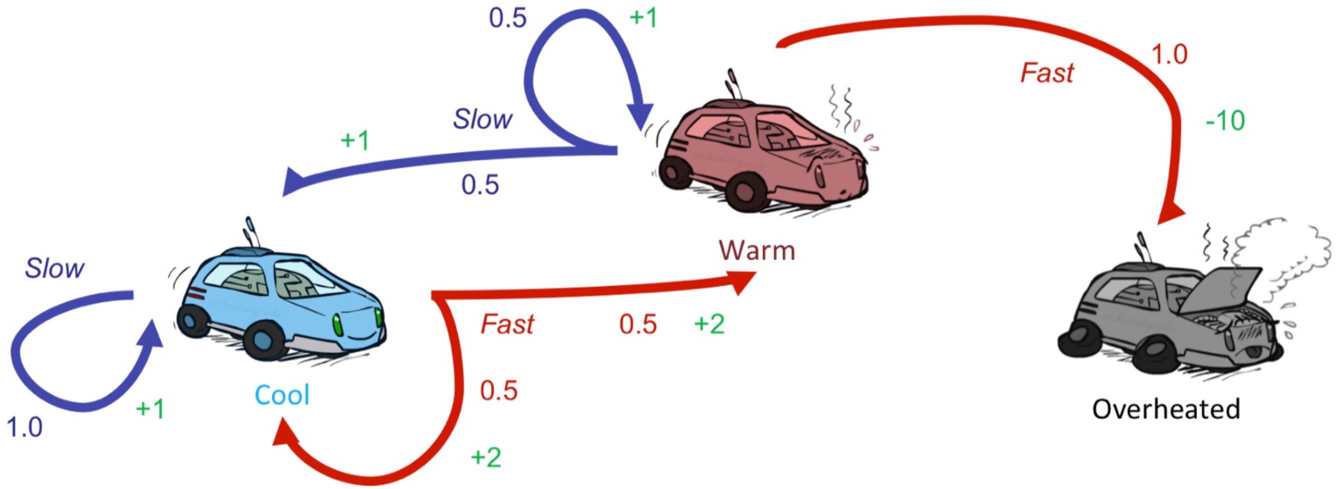
- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees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



Issues with Value Iteration

V_2	2.3 (F)	1.3 (S)	0
V_1	2 (F)	1 (S)	0
V_0	0	0	0

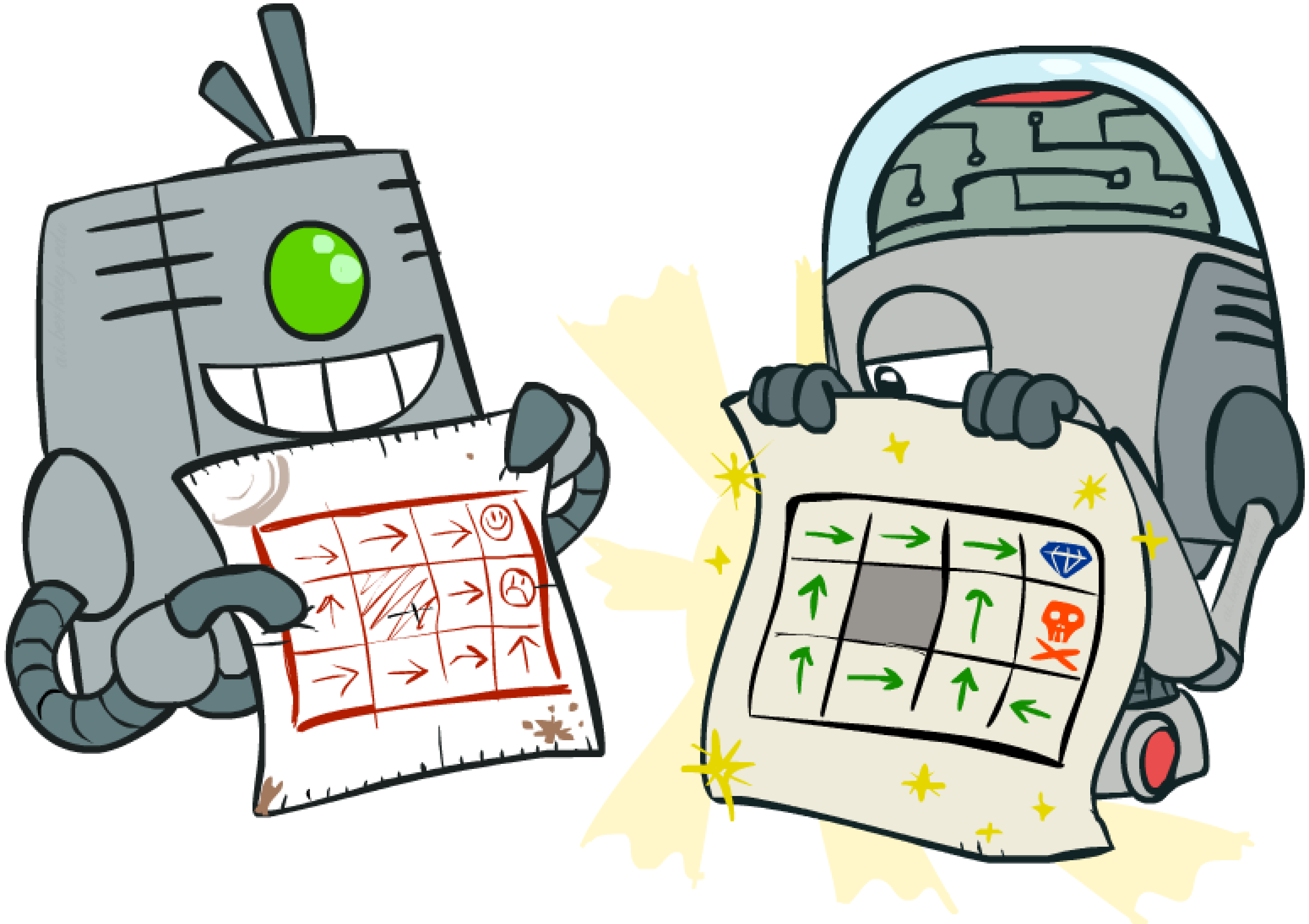
Assuming discount, $\gamma = 0.2$.



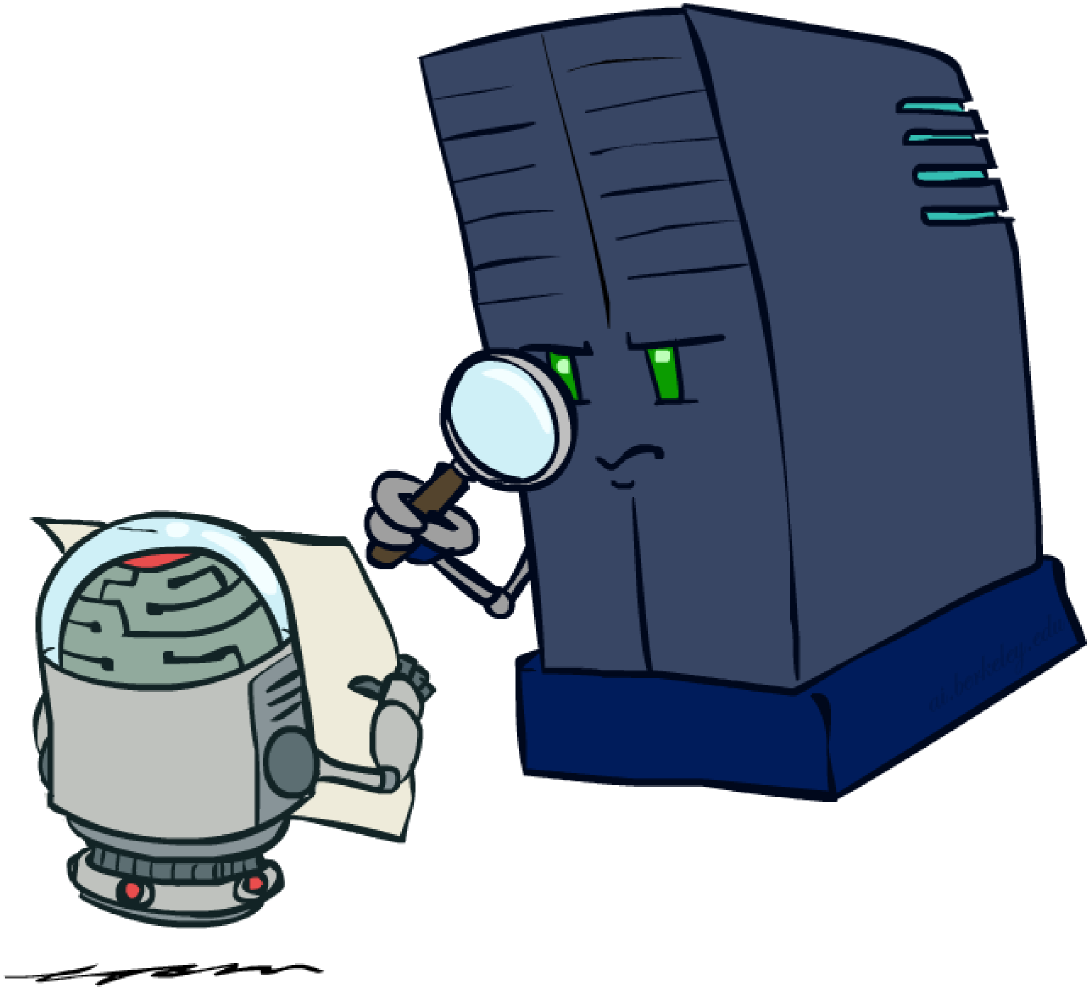
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

- Issue #1: Slow, takes O(S^2A) time.
- Issue #2: Clearly terrible choices keep being tested.
  - e.g. no need to **ever** test the choice: (Warm, Fast).
- Issue #3: Policy converges long before the values.
  - If using value iteration to find best policy, don't actually care about values.

Policy Methods

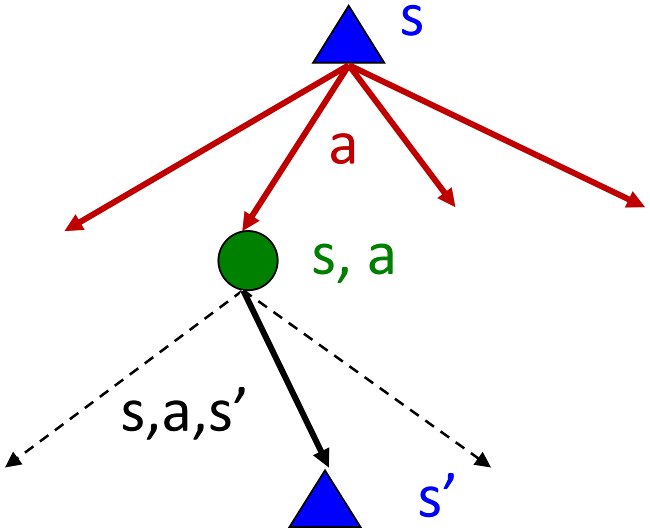


Policy Evaluation

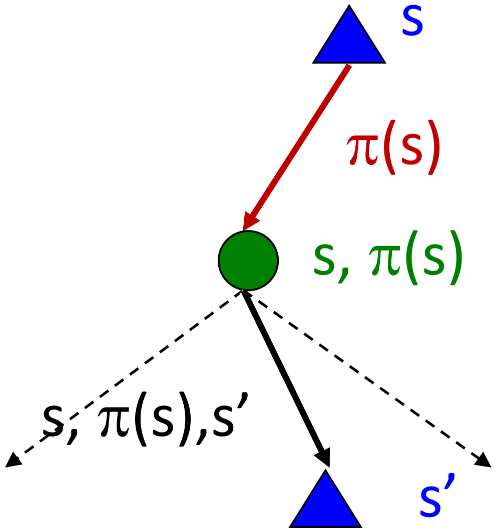


Fixed Policies

Do the optimal action

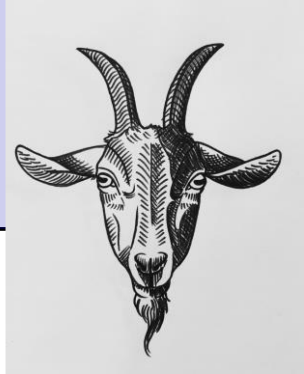


Do what π says to do



- Expectimax trees max over all actions to compute the optimal values
- If we 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

Invent the Bellman Equation (Round 2)

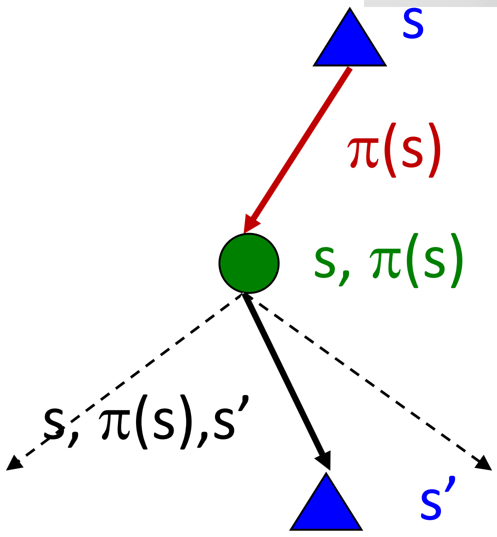


- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- 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

- Challenge: Write the Bellman Equation for $V^{\pi}(s)$



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

Playing optimally

$$V^{\pi}(s) =$$

Playing with π

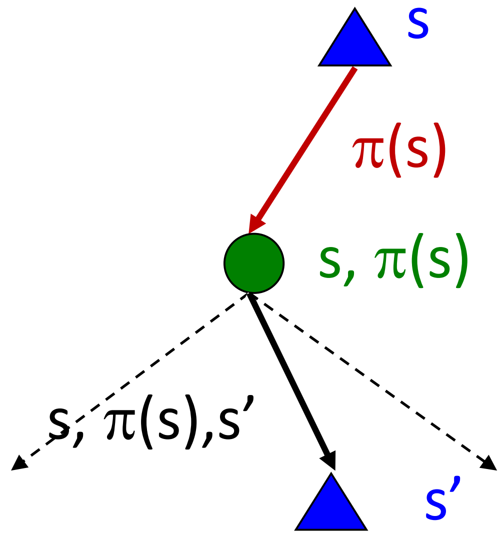
Invent the Bellman Equation (Round 2)

- 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

- Challenge: Write the Bellman Equation for $V^{\pi}(s)$



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

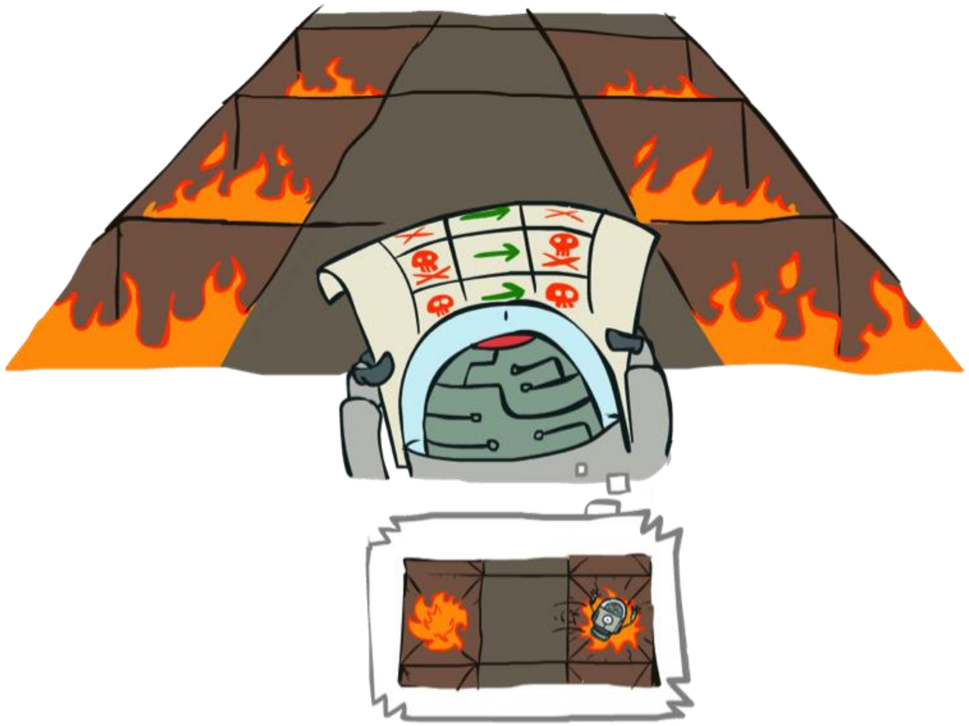
Playing optimally

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

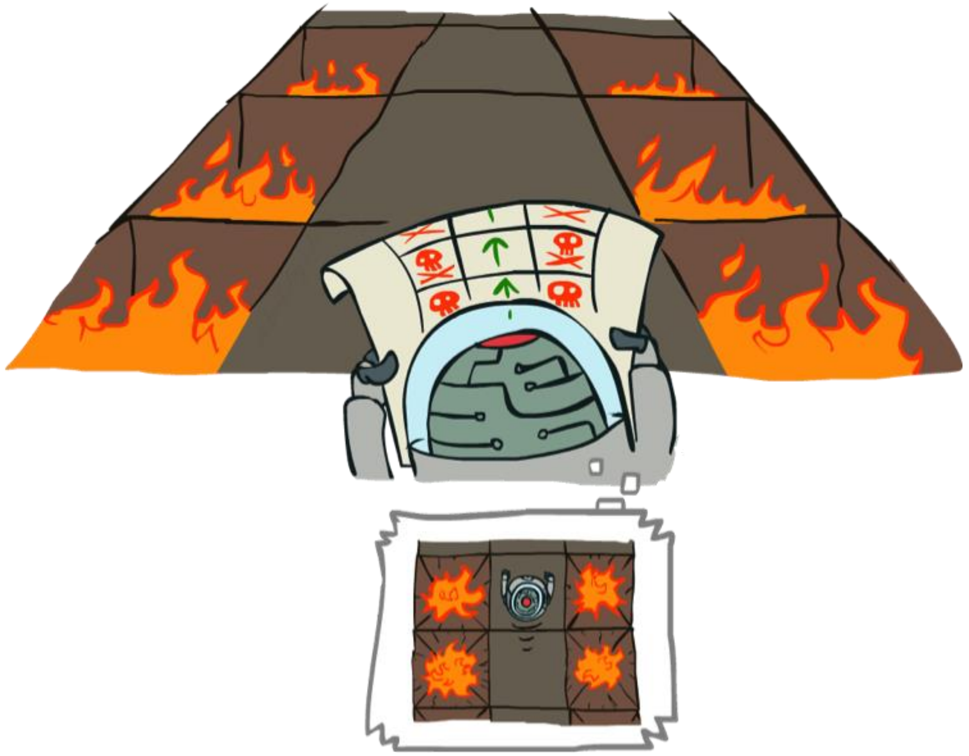
Playing with π

Example: Policy Evaluation

Always Go Right



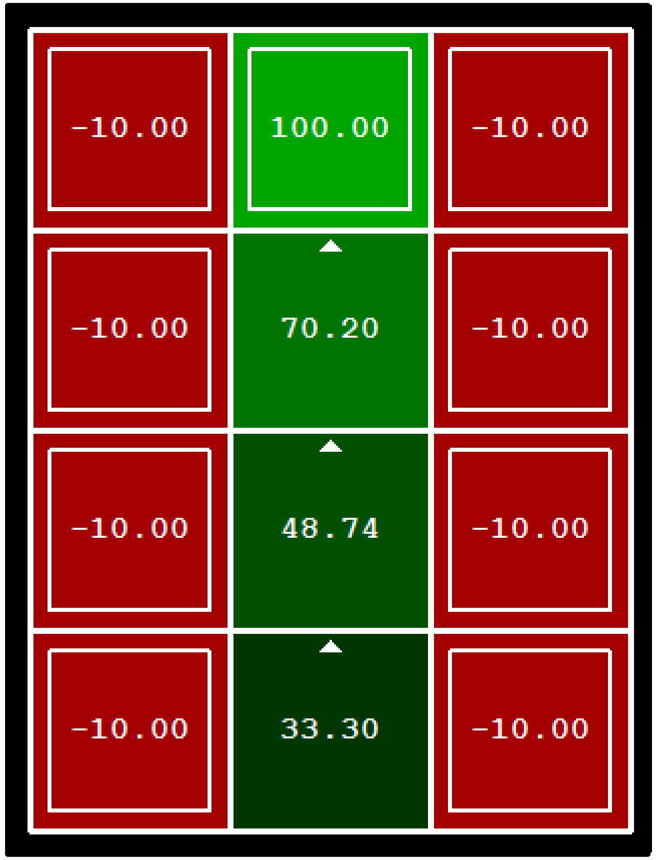
Always Go Forward



Example: Policy Evaluation

Always Go Right

Always Go Forward



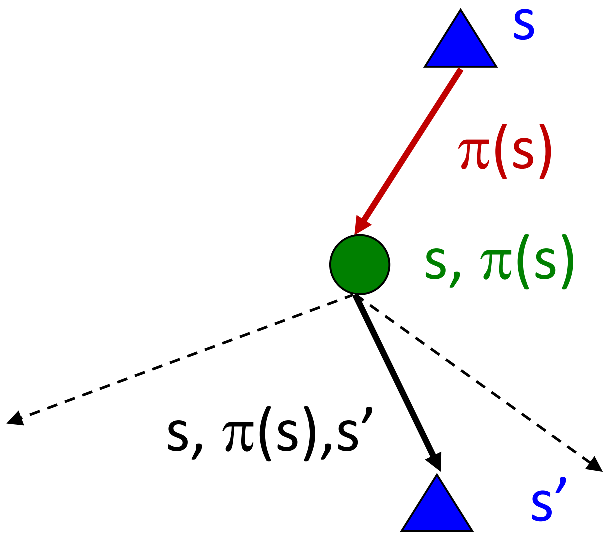
Policy Evaluation

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

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

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

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



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

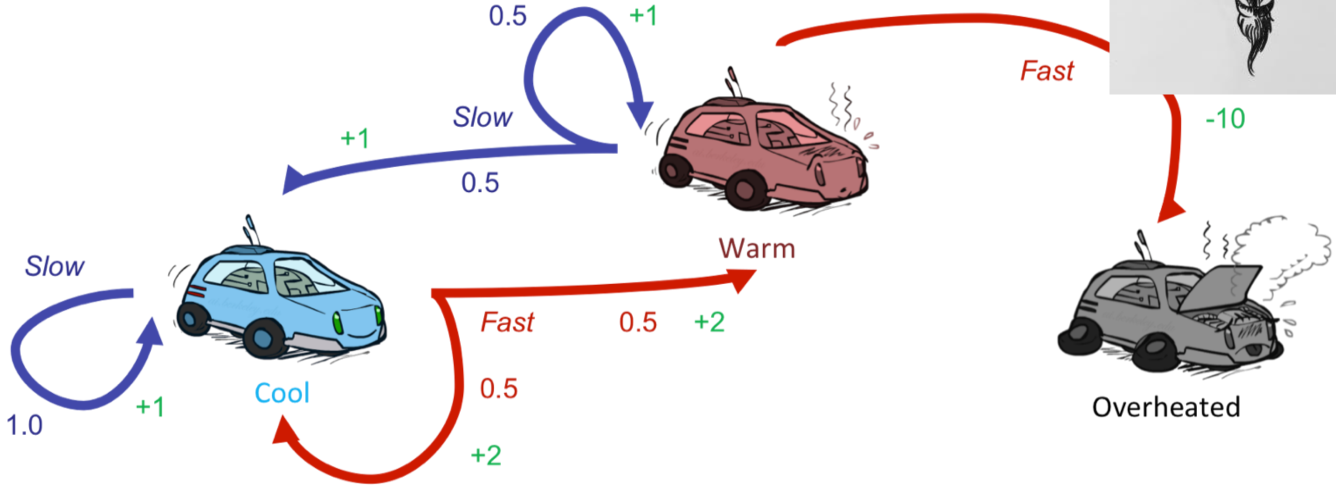
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)

Policy Evaluation Exercise

Evaluate the policy $\pi(s) = Slow$

V_2			
V_1			
V_0	0	0	0

Assume discount $\gamma = 0.2$.



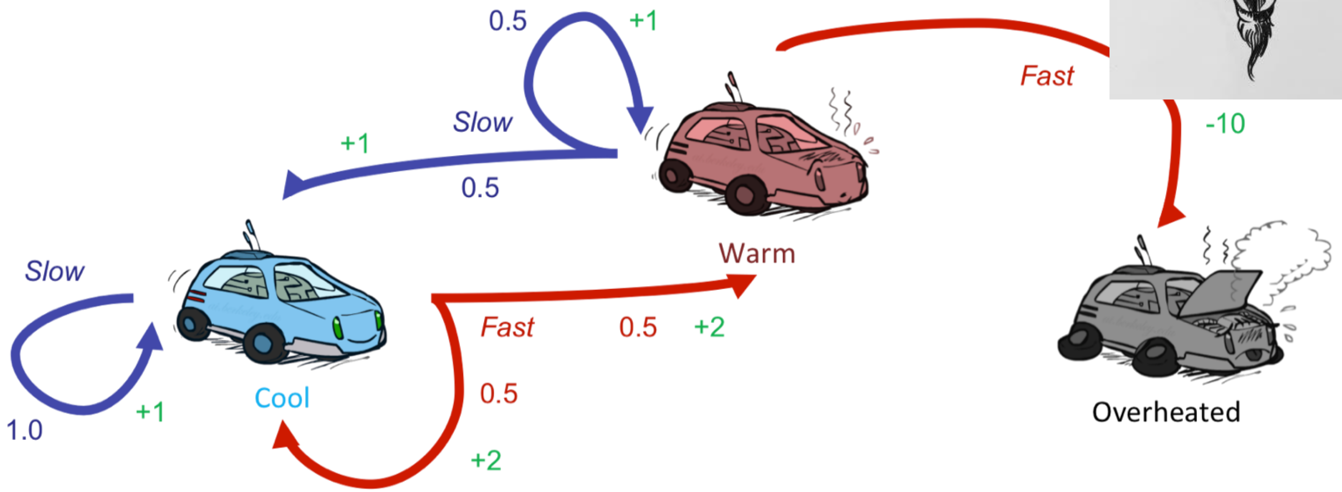
$$V_{k+1}^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

Bonus question, what is $V^{\pi}(s)$?

Policy Evaluation Exercise

Evaluate the policy $\pi(s) = Slow$

V_2	1.2 (S)	1.2 (S)	0
V_1	1 (S)	1 (S)	0
V_0	0	0	0



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

Assume discount $\gamma = 0.2$.

Bonus: Under the given policy, $V^{\pi}(cool)$ and $V^{\pi}(warm)$ converge to 1.25.

Value Iteration vs. Policy Evaluation

- Value iteration lets us compute the expected utility if we play optimally.

0	0	0

$V_0(s)$

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

2	1	0

$V_1(s)$

…

- Policy evaluation lets us compute the expected utility if we play using some policy $\pi(s)$.

0	0	0

$V_0(s)$

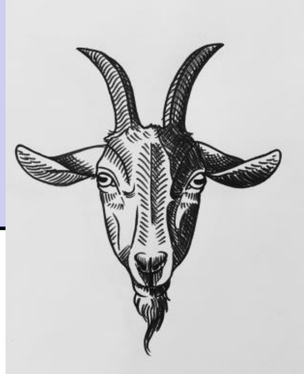
$$V_{k+1}^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')]$$

1	1	0

$V_1(s)$

…

Value Iteration vs. Policy Evaluation



- Value iteration lets us compute the expected utility if we play optimally.

0	0	0

$V_0(s)$

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

2	1	0

$V_1(s)$ …

- Policy evaluation lets us compute the expected utility if we play using some policy $\pi(s)$.

0	0	0

$V_0(s)$

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

1	1	0

$V_1(s)$ …

- Let **X(s)** be the vector resulting from running policy evaluation on the **optimal policy**.
- Let **Y(s)** be the vector resulting from running policy evaluation on **a policy** which always selects the “leftmost” action, i.e. first action on the list of possible actions.
- Let **Z(s)** be the vector resulting from running **value iteration**.

Assuming we run them all to convergence, how are the values of X, Y, and Z related?

Value Iteration vs. Policy Evaluation

- Value iteration lets us compute the expected utility if we play optimally.

$$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}_{V_0(s)} \longrightarrow \boxed{V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]} \longrightarrow \begin{bmatrix} 2 & 1 & 0 \end{bmatrix}_{V_1(s)} \xrightarrow{\cdots}$$

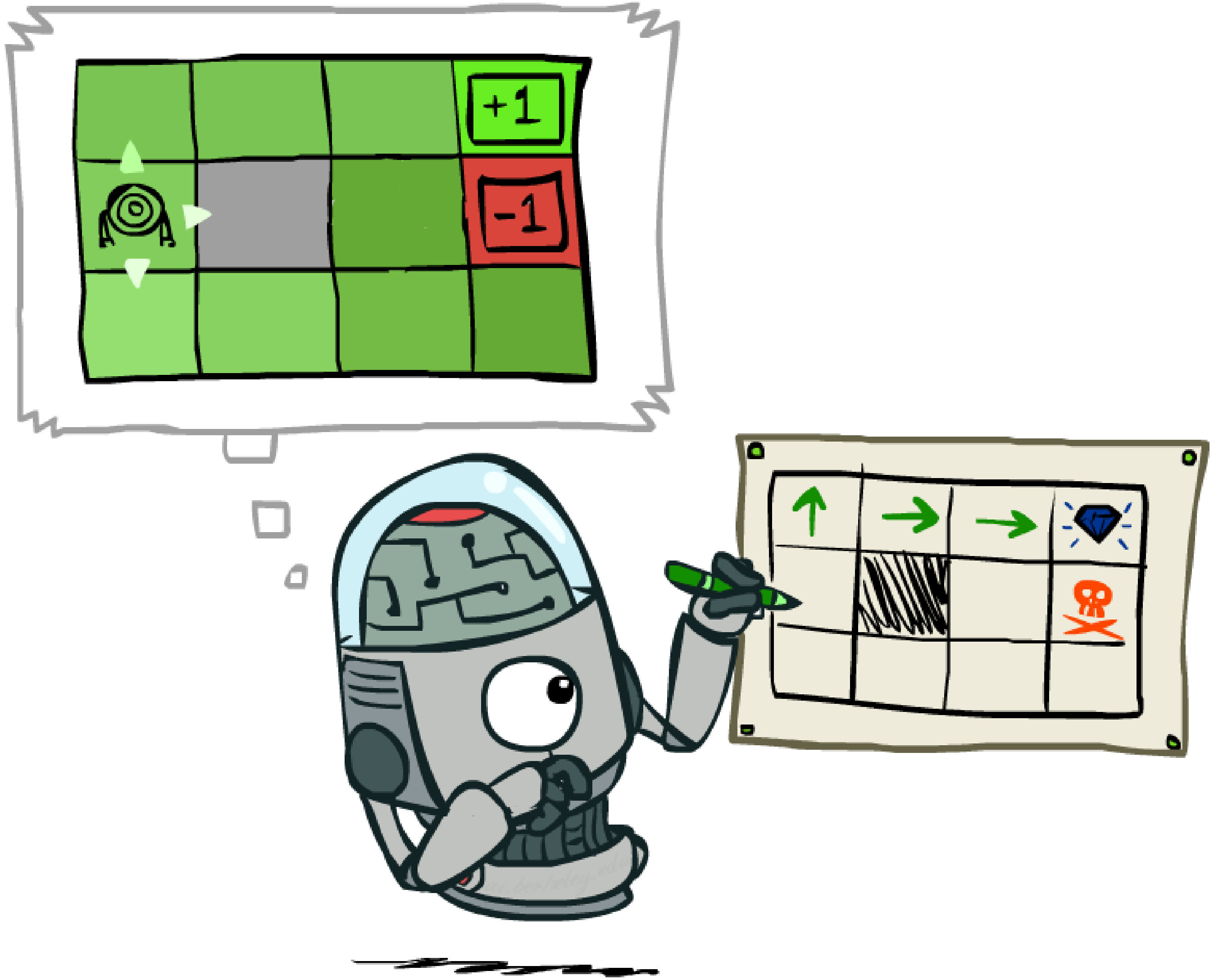
- Policy evaluation lets us compute the expected utility if we play using some policy $\pi(s)$.

$$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}_{V_0(s)} \longrightarrow \boxed{V^{\pi}_{k+1}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}_k(s')]} \longrightarrow \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}_{V_1(s)} \xrightarrow{\cdots}$$

- Let **X(s)** be the vector resulting from running **policy evaluation** on the **optimal policy**.
- Let **Y(s)** be the vector resulting from running **policy evaluation** on **a policy** which always selects the "leftmost" action, i.e. first action on the list of possible actions.
- Let **Z(s)** be the vector resulting from running **value iteration**.

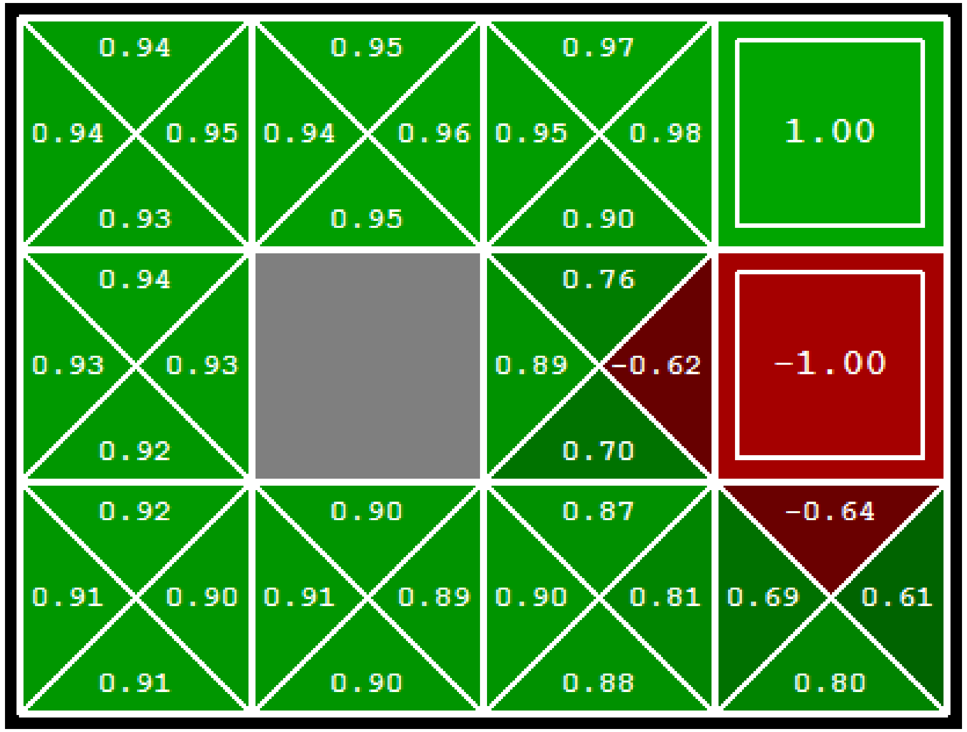
Answer: X and Z are equal. All values in Y are less than or equal to X.

Policy Extraction



Computing Actions from Q-Values

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



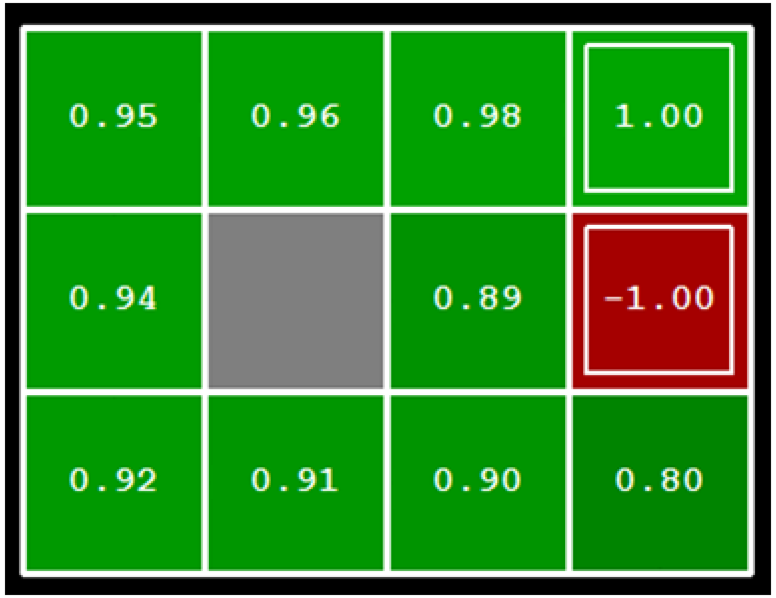
- What is the optimal policy?
  - Completely trivial to decide!

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

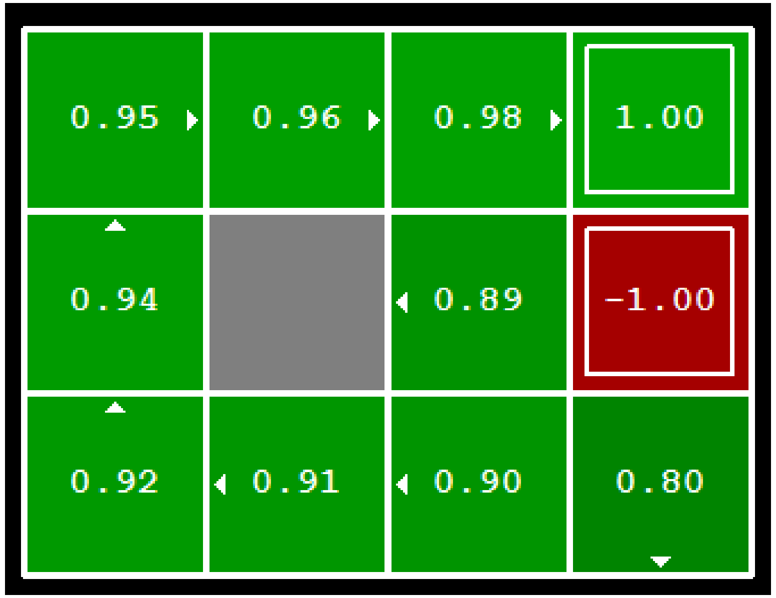
- Important lesson: actions are easier to select from q-values than values!

Computing Actions from Values

- Let's imagine we have the optimal values V*(s)
- What is the optimal policy, i.e. how should we act?
  - It's not obvious!

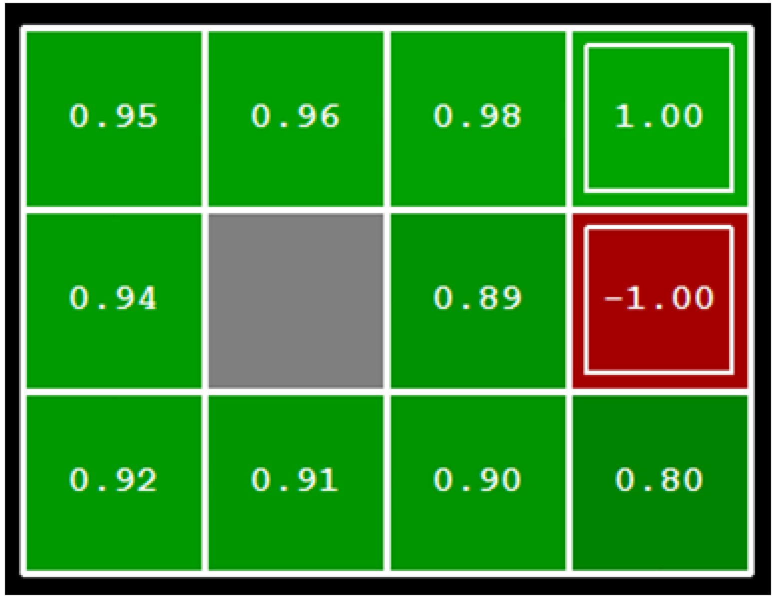


???



Computing Actions from Values

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



- What is the optimal policy?
  - It's not obvious!
- We need to do a mini-expectimax (one step)
  - In other words, compute q-values, pick action that goes with maximum for each state.

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

- This is called policy extraction, since it gets the policy implied by the values
  - Important observation: actions are easier to select from q-values than values!

Value Iteration, Policy Evaluation, Policy Extraction

- Value iteration lets us compute the expected utility if we play optimally.

0	0	0

$V_0(s)$

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

2	1	0

$V_1(s)$ …

- Policy evaluation lets us compute the expected utility if we play using some policy $\pi(s)$.

0	0	0

$V_0(s)$

$$V^{\pi}_{k+1}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}_k(s')]$$

1	1	0

$V_1(s)$ …

- Policy extraction lets us compute the optimal policy from an optimal vector $V^*(s)$.

2.375	1.375	0

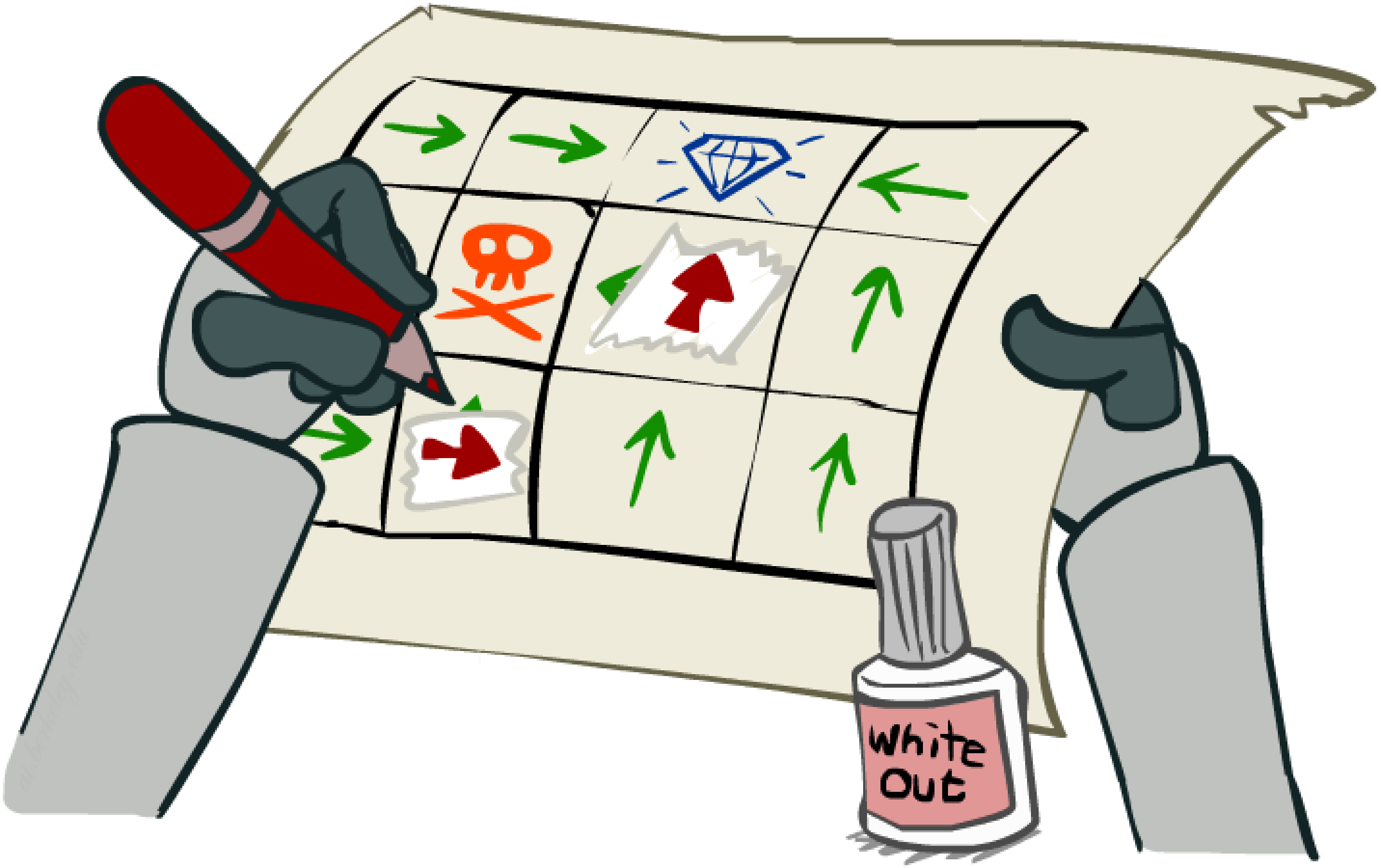
$V^*(s)$

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

Fast	Slow	Ø

$\pi^*(s)$

Policy Iteration



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

0	0	0

$V_0(s)$

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

3.5	2.6	0

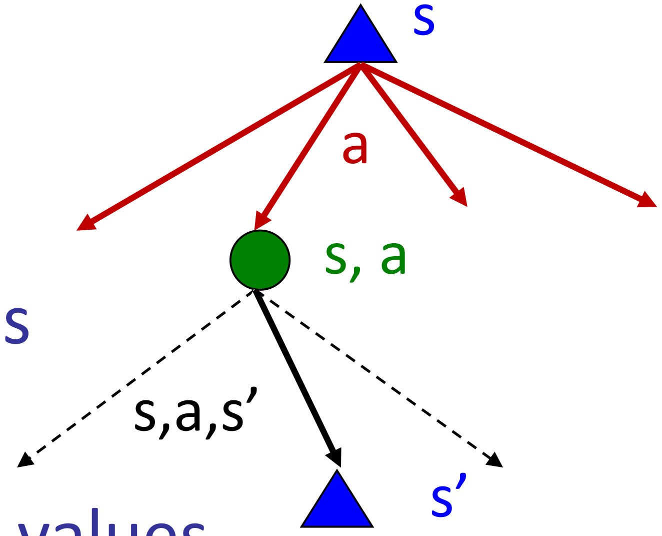
$V_1(s)$

…

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

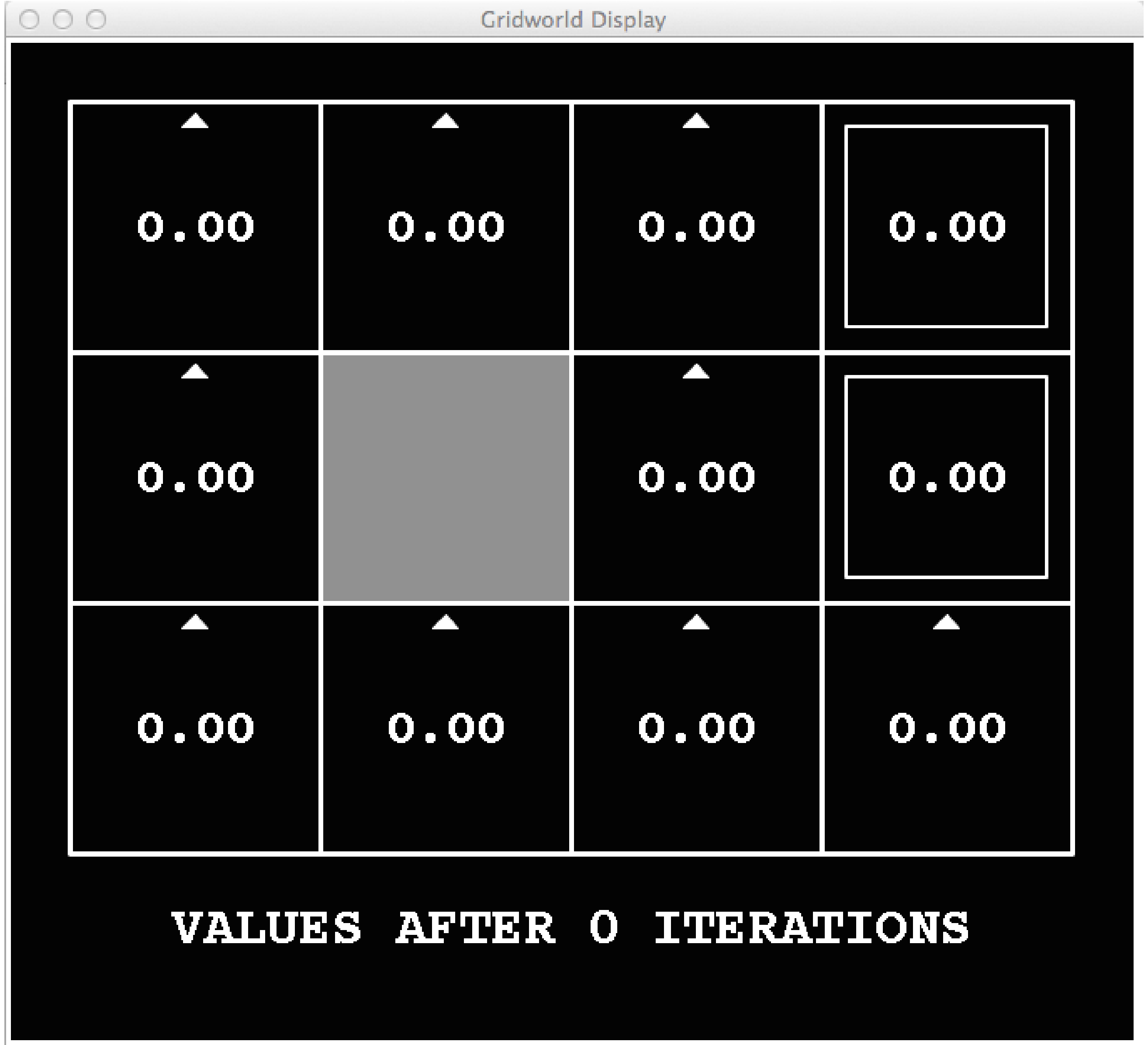
- Problem 2: The best action at each state rarely changes

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



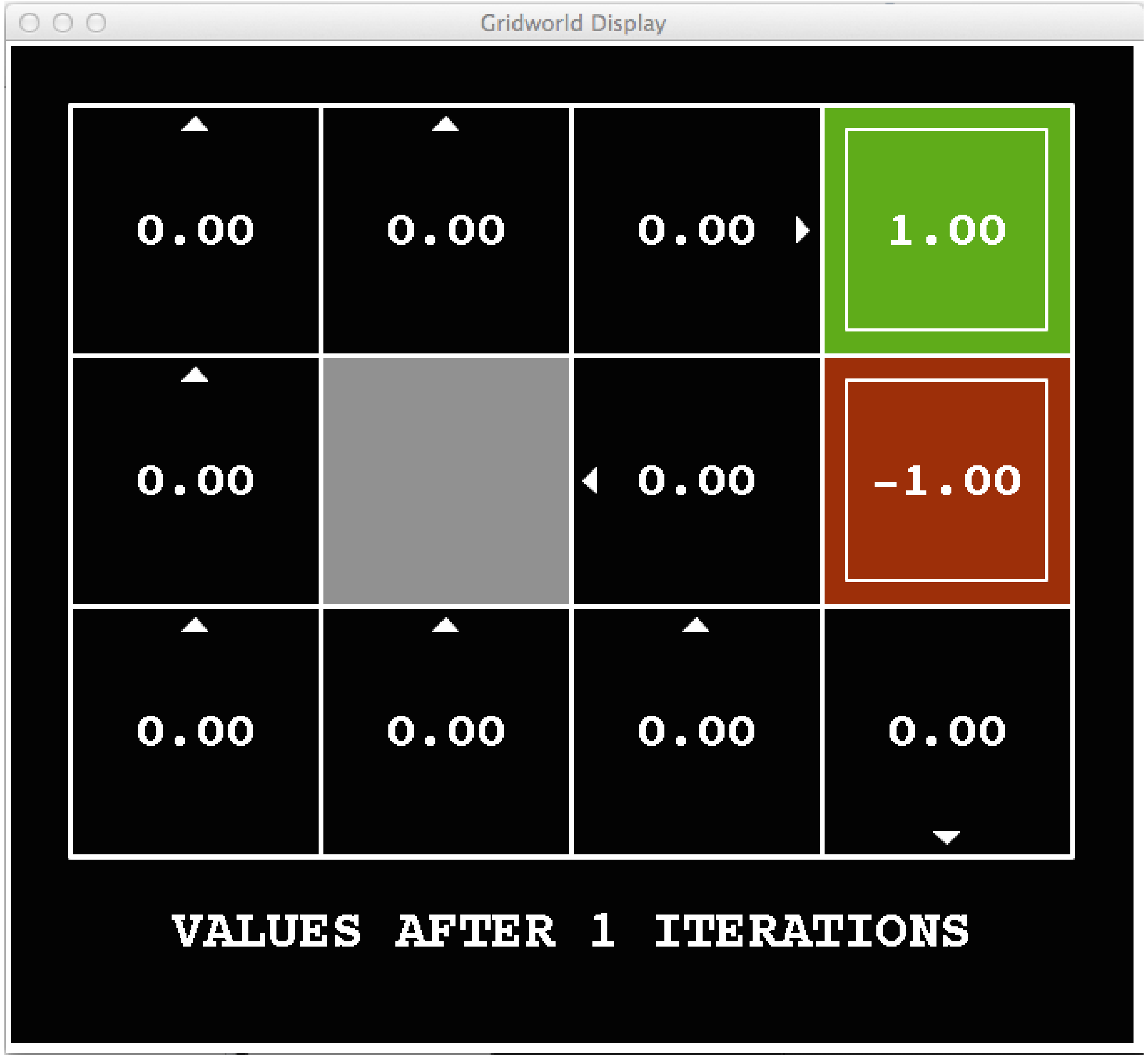
[Demo: value iteration (L9D2)]

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

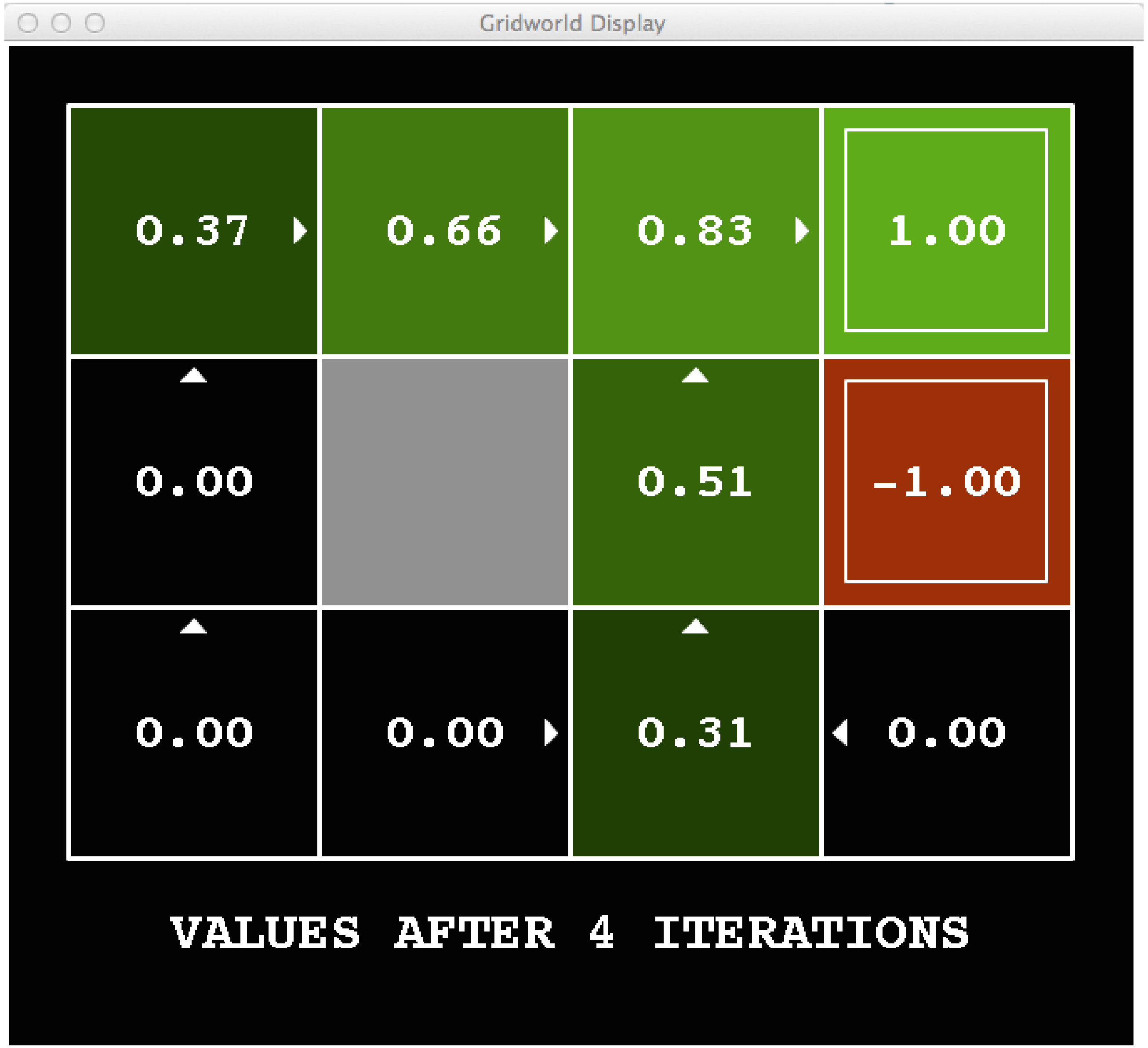
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

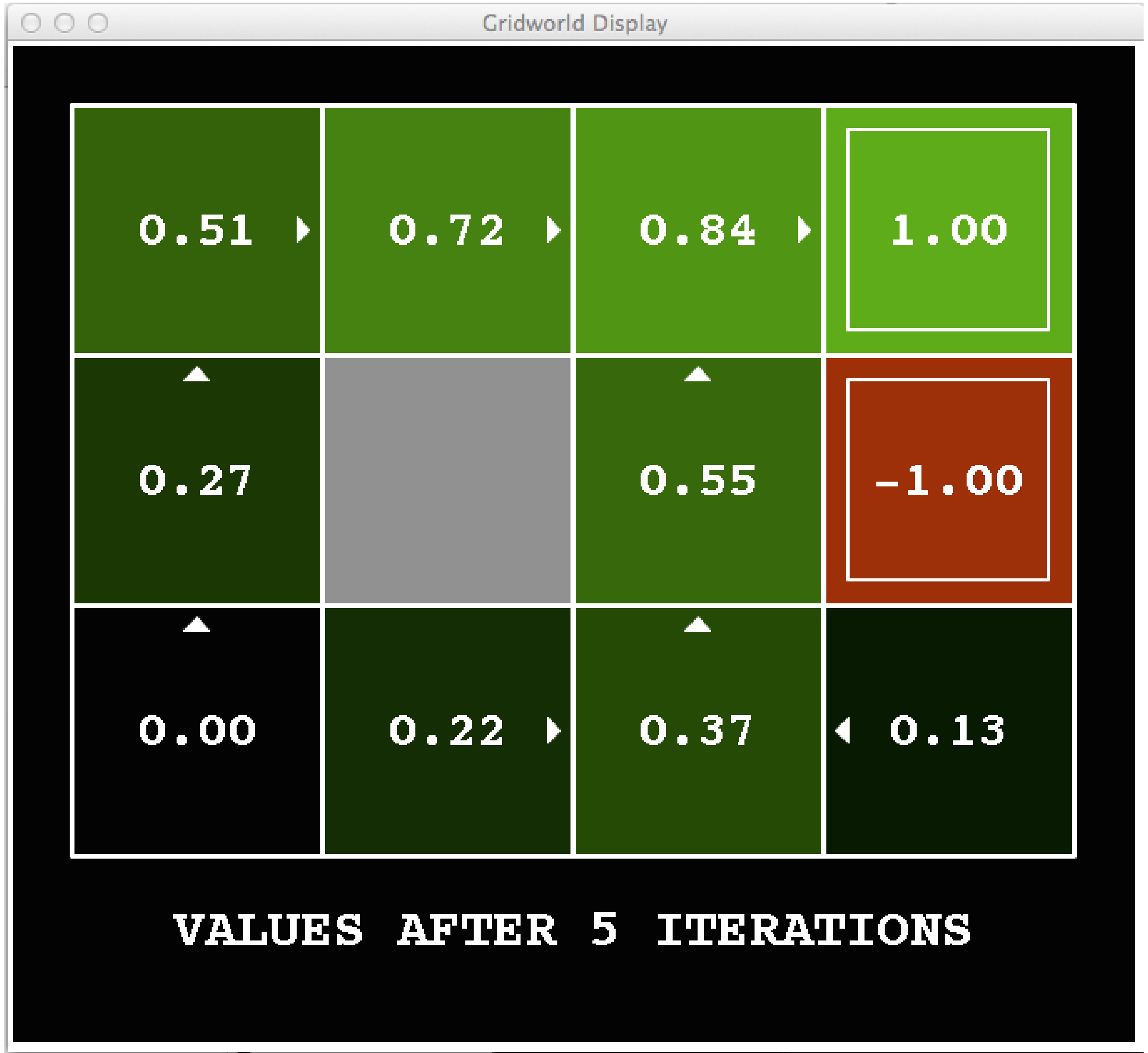
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



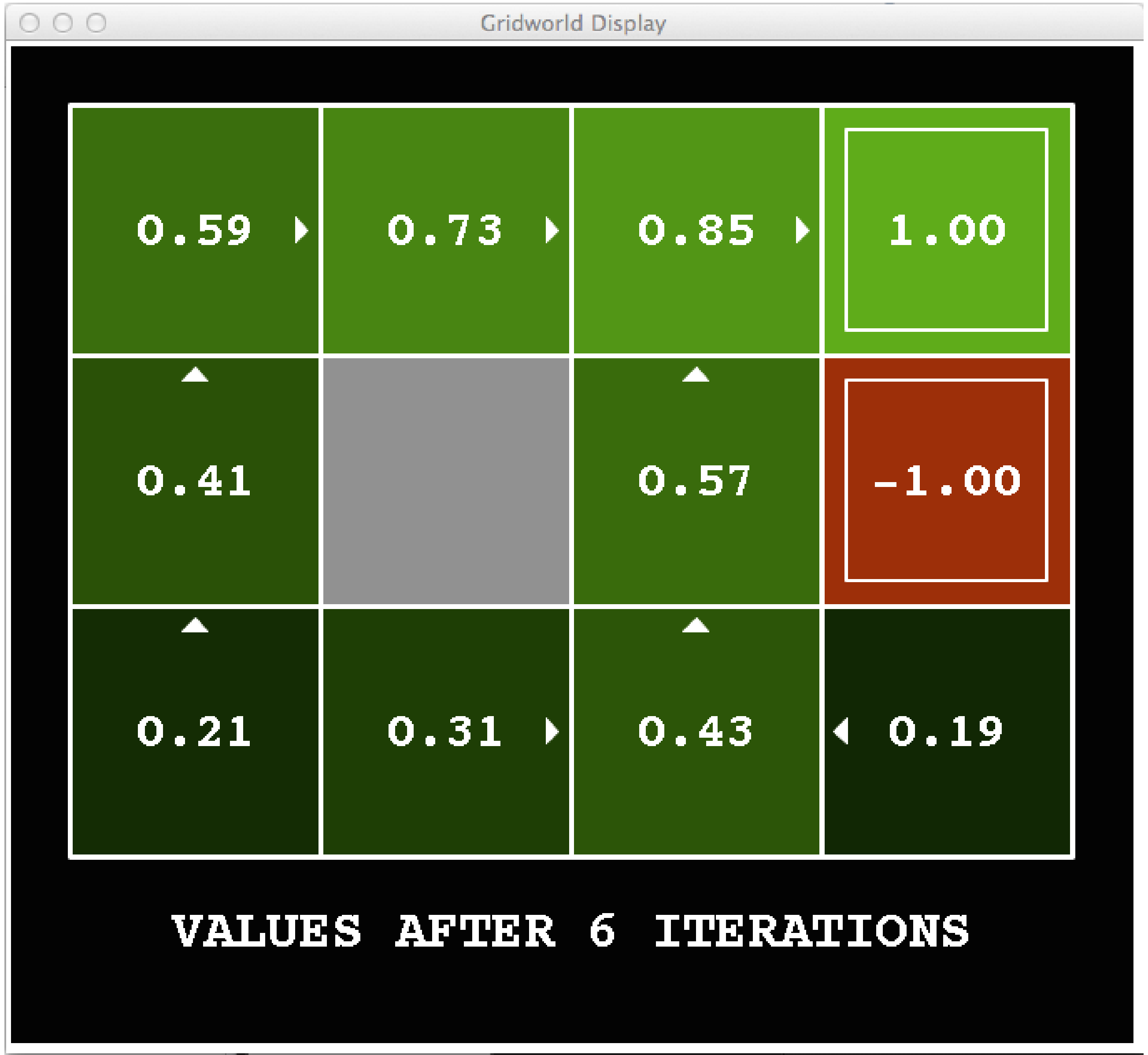
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



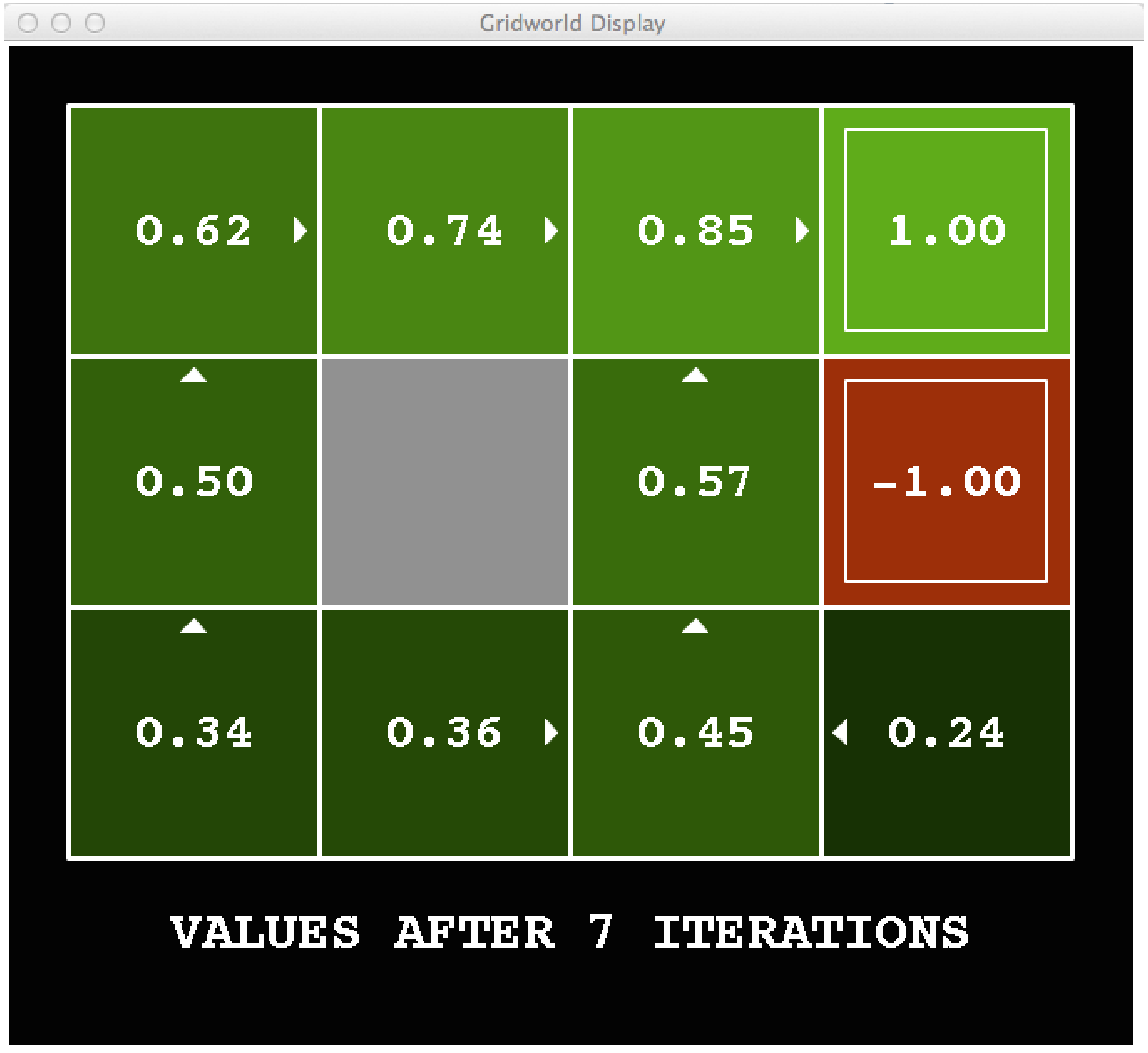
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



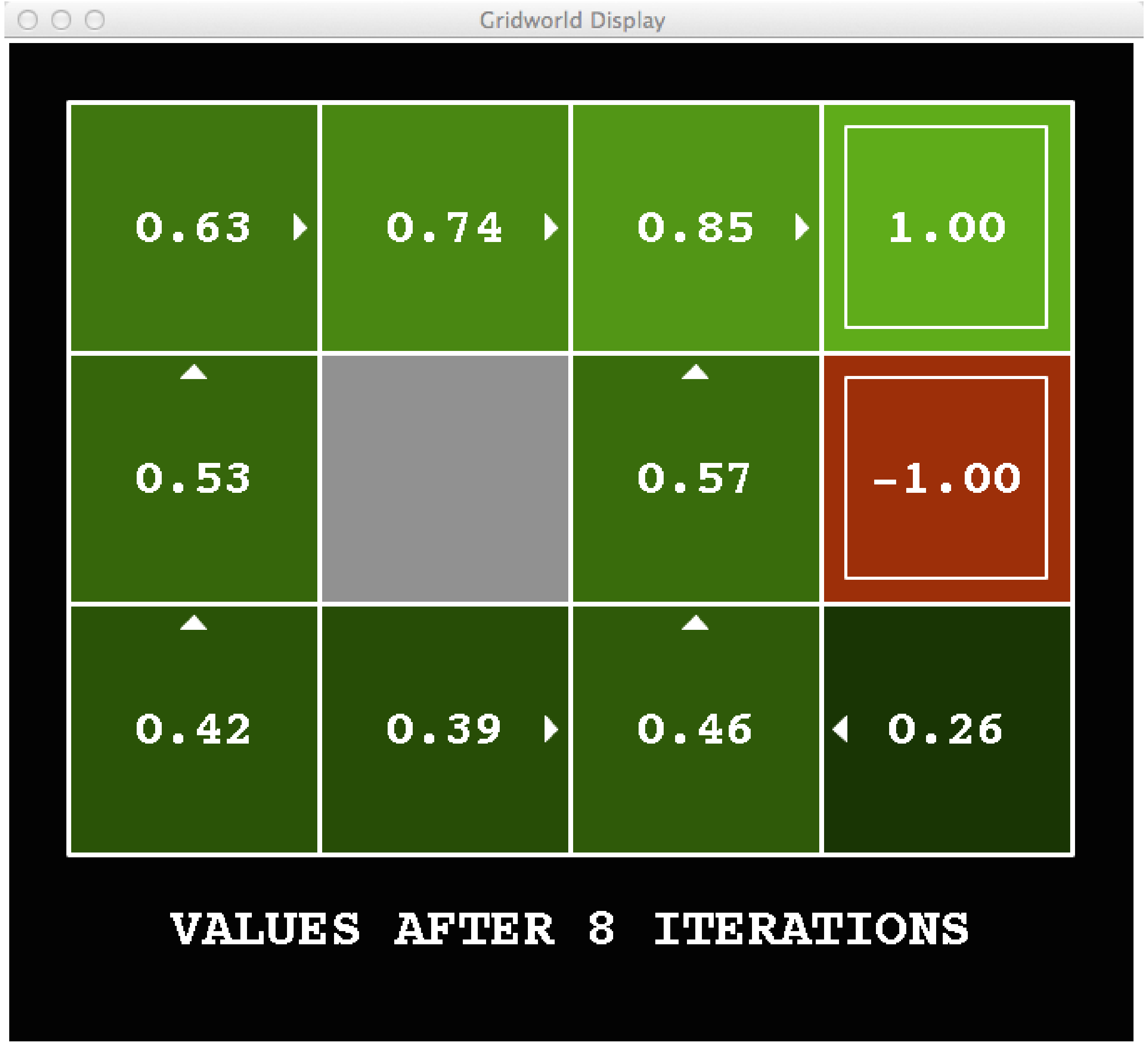
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



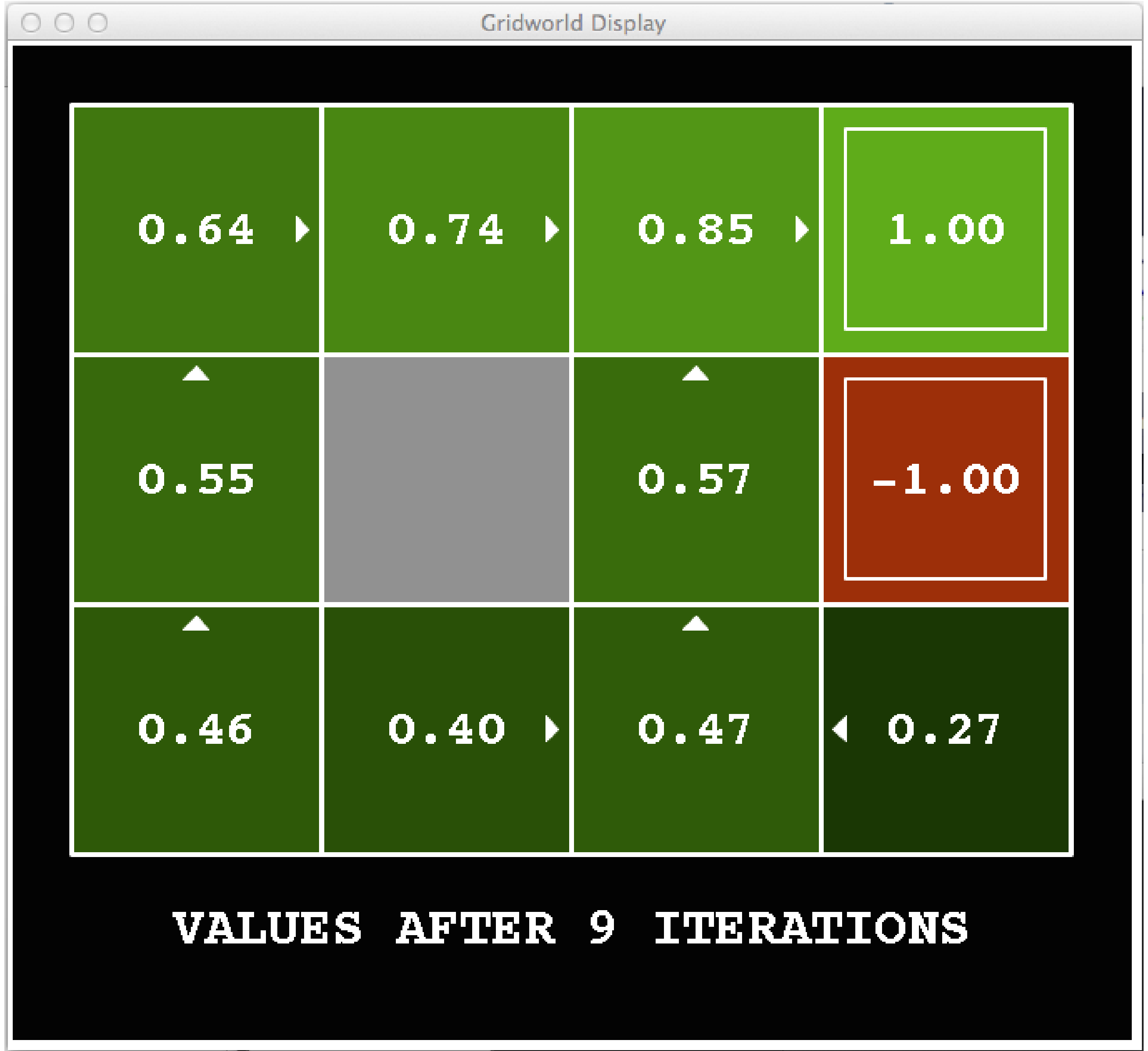
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



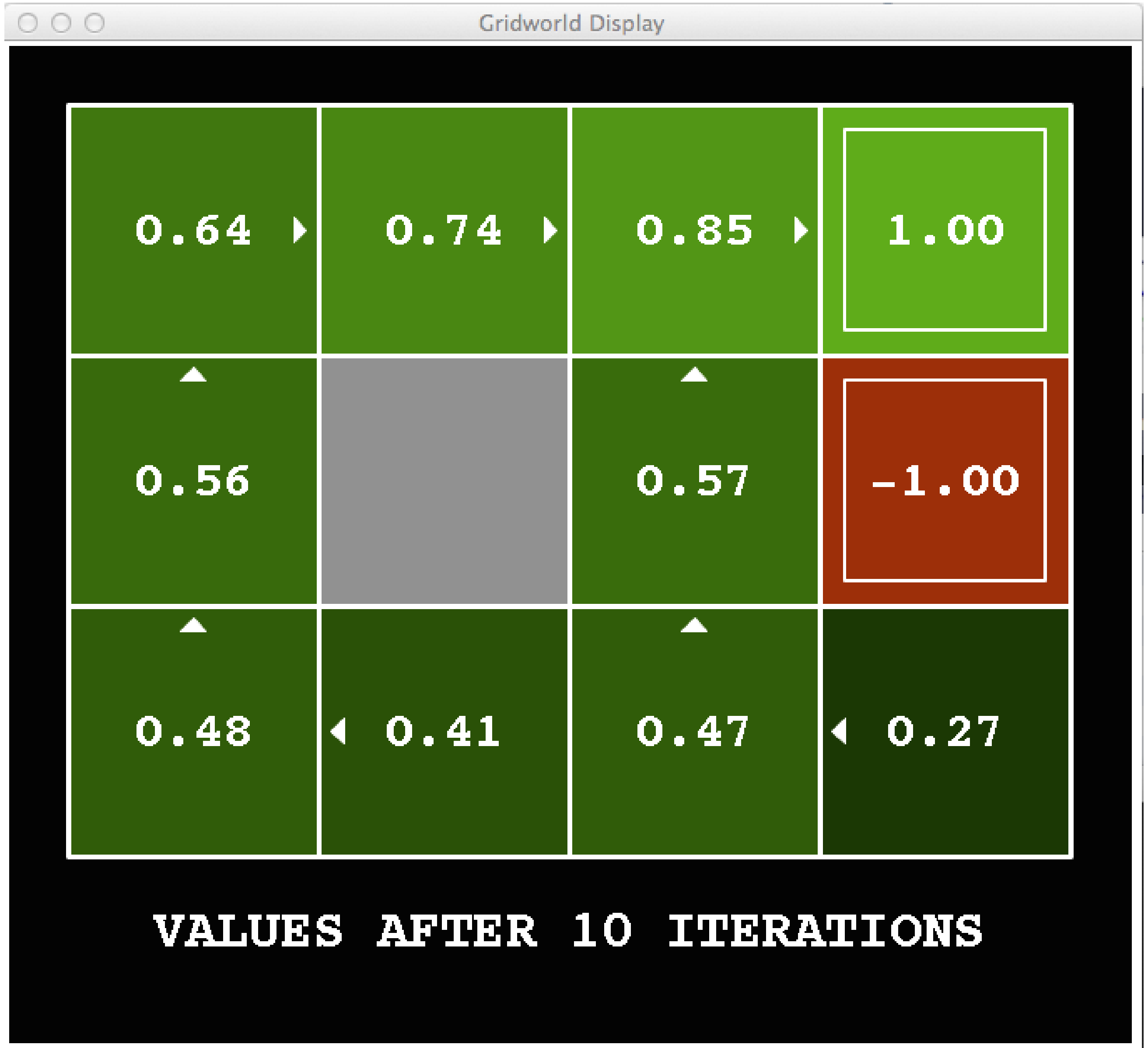
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



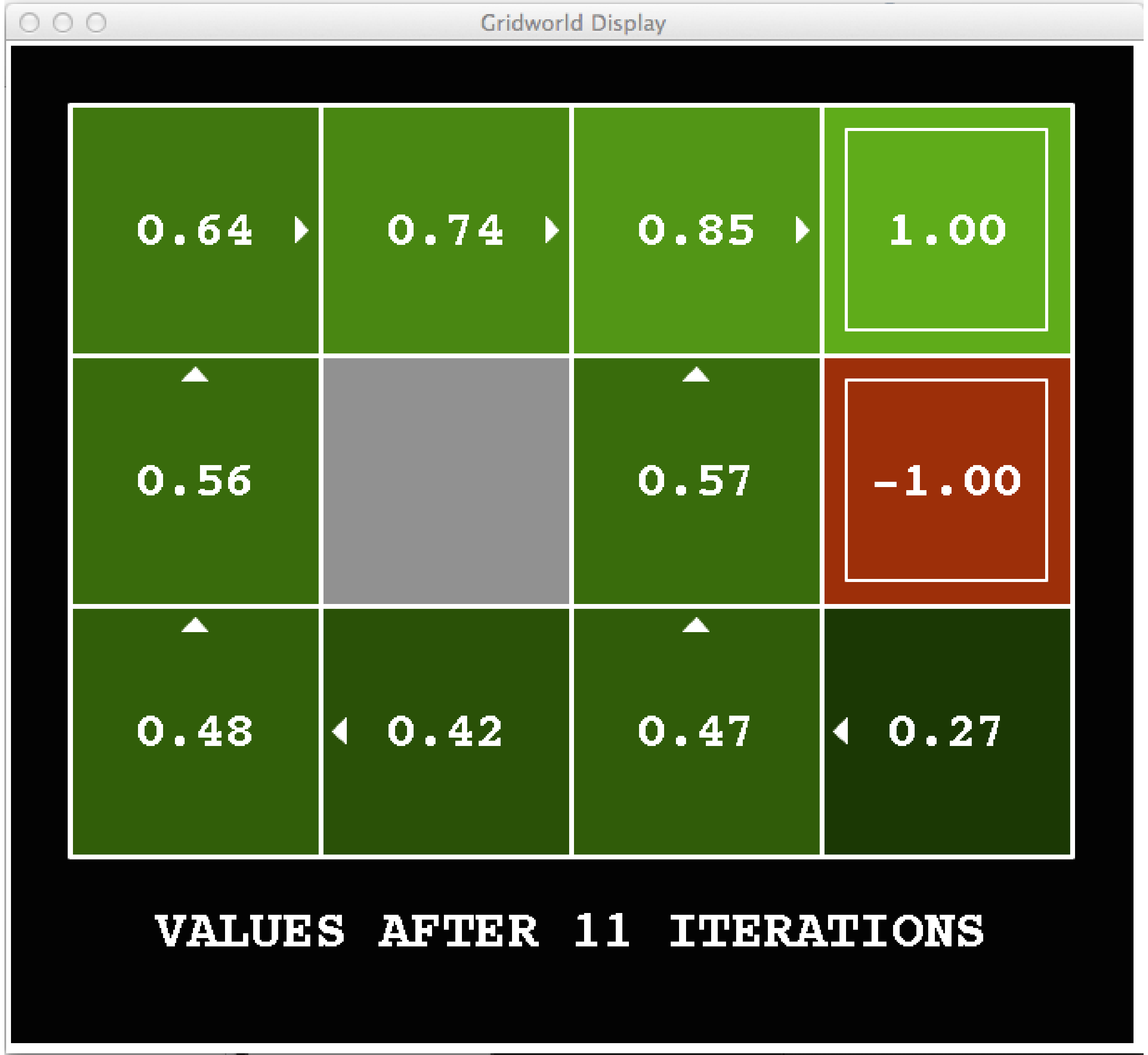
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



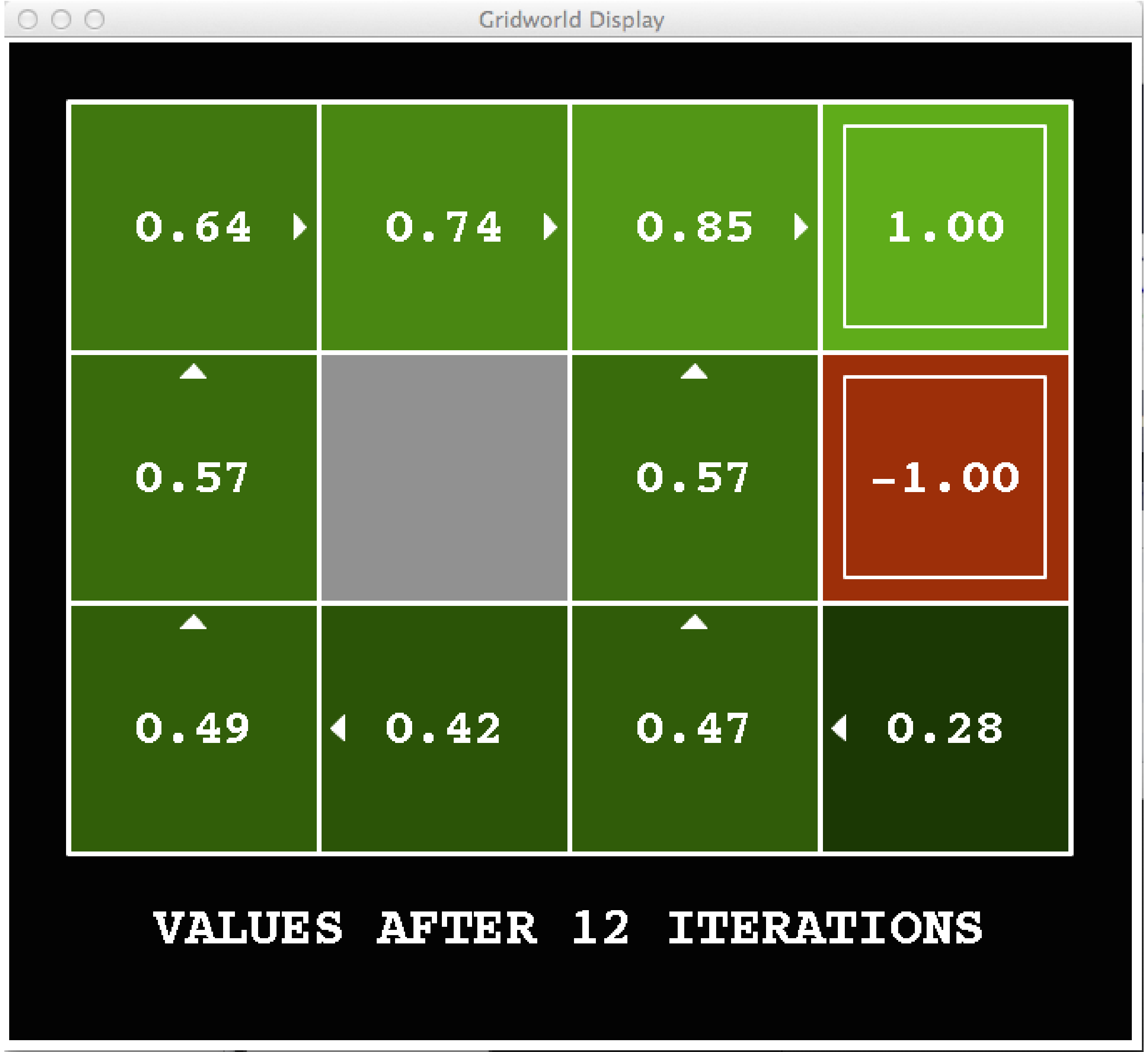
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



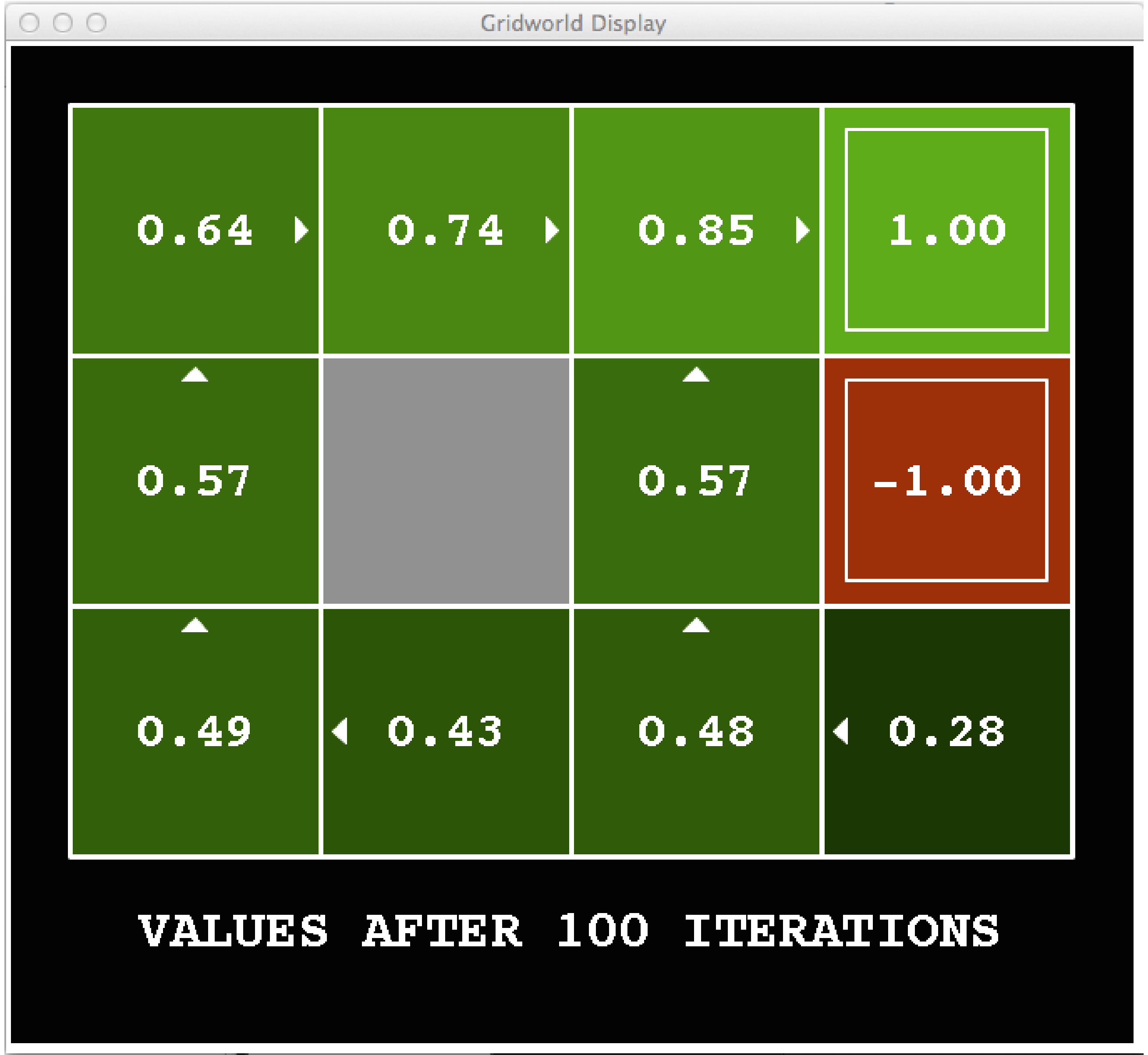
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

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

- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

Policy Iteration

- Evaluation: For fixed current policy π_i, find values with **policy evaluation**:
  - Iterate until values converge:

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

- Improvement: For fixed values, get a better policy using **policy extraction**
  - Do a one-step look ahead using expectimax.
  - In other words, compute q-values, pick action that goes with maximum for each state.

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

Policy Evaluation, Extraction, and Iteration

- Policy **evaluation** lets us compute the expected utility if we play using some policy $\pi(s)$.

0	0	0

$V_0(s)$

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

1	1	0

$V_1(s)$

...

- Policy **extraction** lets us compute the optimal policy from an optimal vector $V^*(s)$.

2.375	1.375	0

$V^*(s)$

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

Fast	Slow	Ø

$\pi^*(s)$

- Policy iteration is a call to **evaluate**, followed by a call to **extract**.
  - Given a policy π_i, policy iteration outputs a better policy π_{i+1}.

Slow	Slow	Ø

π_0

V^{π_0} = **evaluate**(π_0)

π_1 = **extract**(V^{π_0})

iterate

Fast	Slow	Ø

π_1

More Thorough Example

- Come up with an **arbitrary initial policy** π_0, say $\pi_0(s)$ = [slow, slow, Ø].
- Compute the expected utilities for this policy if we use policy π_0 using **policy evaluation**.
  - After converging (possibly many many iterations), this yields a vector $V^{\pi_i}(s)$.

$$\pi_0(s) = [\text{slow}, \text{slow}, \emptyset]$$

0	0	0

$V_0^{\pi_0}(s)$

$$V_1^{\pi_0}(s) = \sum_{s'} T(s,\pi_0(s),s')[R(s,\pi_0(s),s') + \gamma V_0^{\pi_0}(s')]$$

1	1	0

$V_1^{\pi_0}(s)$

1	1	0

$V_1^{\pi_0}(s)$

$$V_2^{\pi_0}(s) = \sum_{s'} T(s,\pi_0(s),s')[R(s,\pi_0(s),s') + \gamma V_1^{\pi_0}(s')]$$

1.2	1.2	0

$V_2^{\pi_0}(s)$

...

$V_{182}^{\pi_0}(s)$

1.25	1.25	0

More Thorough Example

- Come up with an **arbitrary initial policy** π_0, say $\pi_0(s)$ = [slow, slow, Ø].
- Compute the expected utilities for this policy if we use policy π_0 using **policy evaluation**.
  - After converging (possibly many many iterations), this yields a vector $V^{\pi_i}(s)$.
  - The expected utility of the policy [slow, slow, Ø] is [1.25, 1.25, 0]
- Compute a better policy π_1 using **policy iteration**.
  - In other words, compute q-values for each possible choice, pick new best.

1.25	1.25	0

$V_{182}^{\pi_0}(s)$

1.25	1.25	0

$V^{\pi_0}(s)$

$$\pi_1(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{\pi_0}(s')]$$

Fast	Slow	Ø

$\pi_1(s)$

- Can repeat this process to get even better policies π_2, π_3, etc.

Value Iteration vs. Policy Iteration

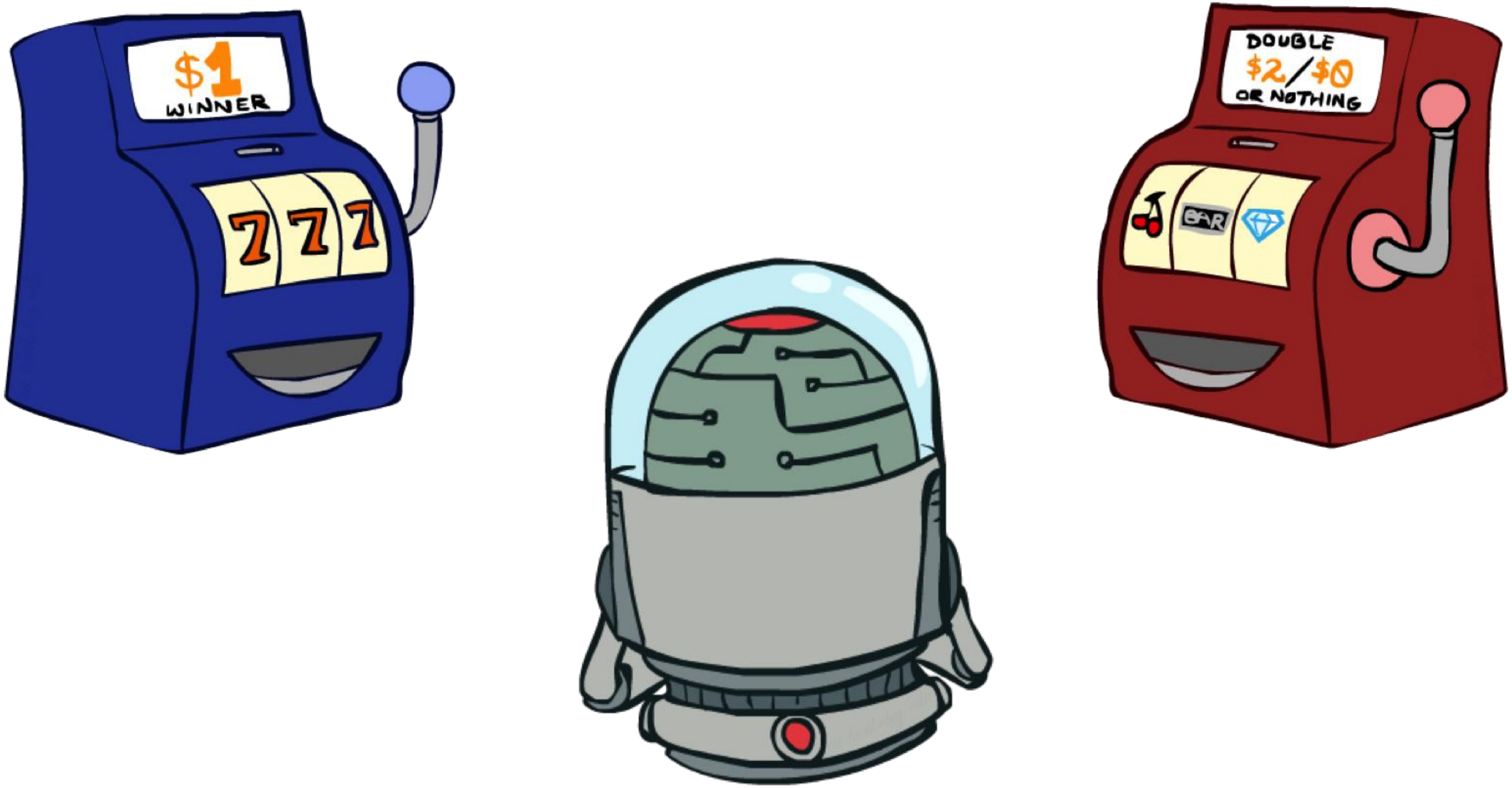
- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration (previous lecture):
  - Every iteration updates both the values and (implicitly) the policy
  - We don’t track the policy, but taking the max over actions implicitly recomputes it
- In policy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

Summary: MDP Algorithms

- So you want to….
  - Compute optimal values: use **value iteration** or **policy iteration**
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions

Double Bandits



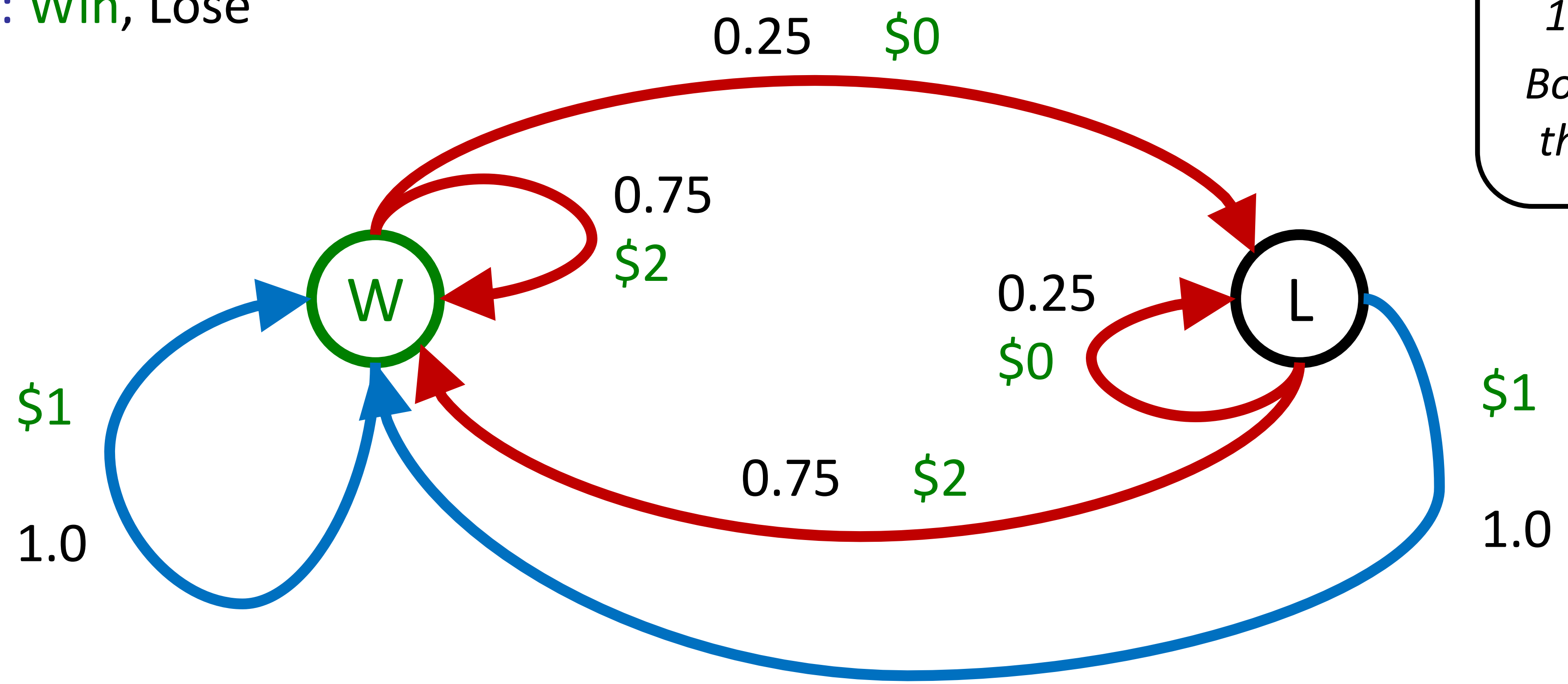
Double-Bandit MDP

- Actions: *Blue*, *Red*
- States: Win, Lose

No discount

100 time steps

Both states have the same value



Offline Planning

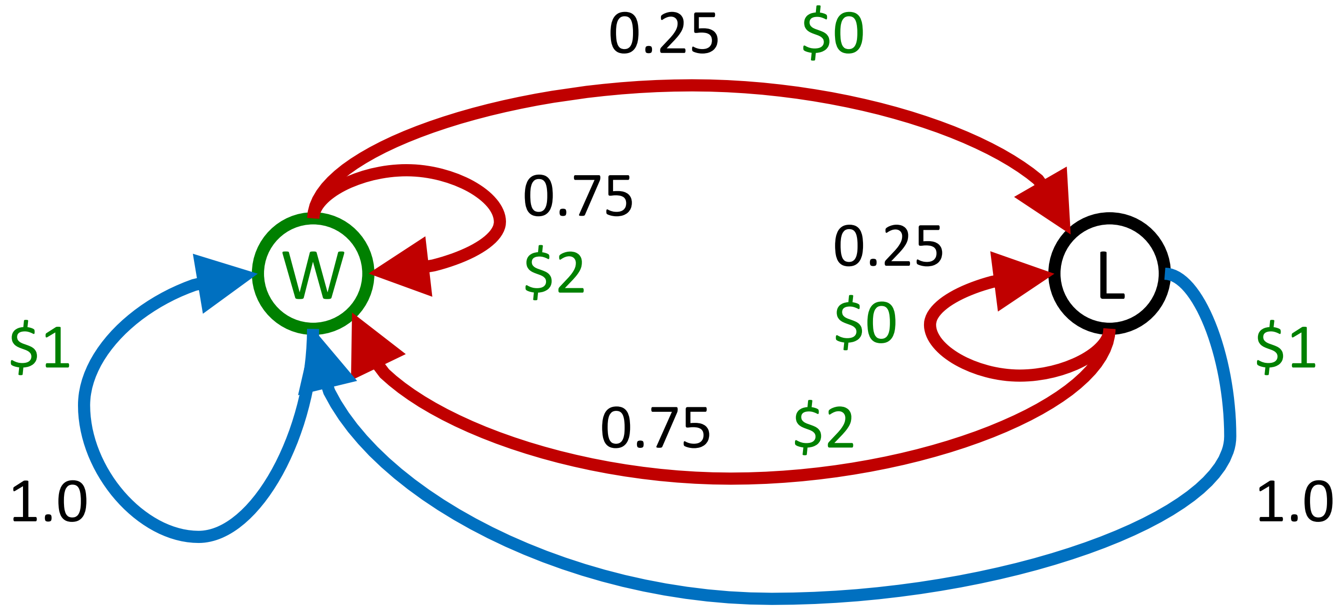
- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

No discount

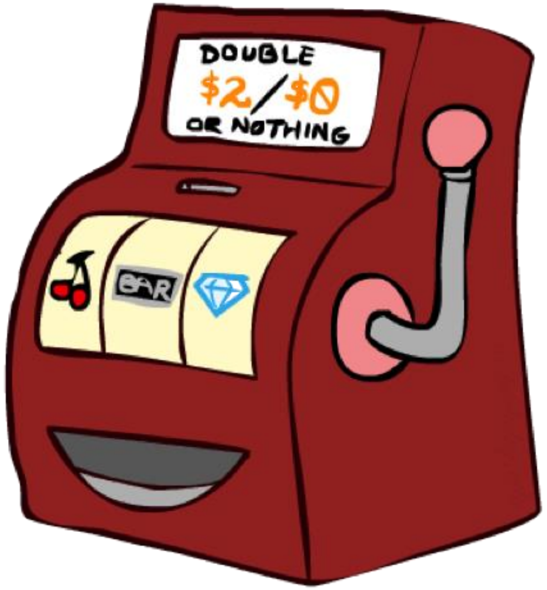
100 time steps

Both states have the same value

	Value
Play Red	150
Play Blue	100



Let’s Play!

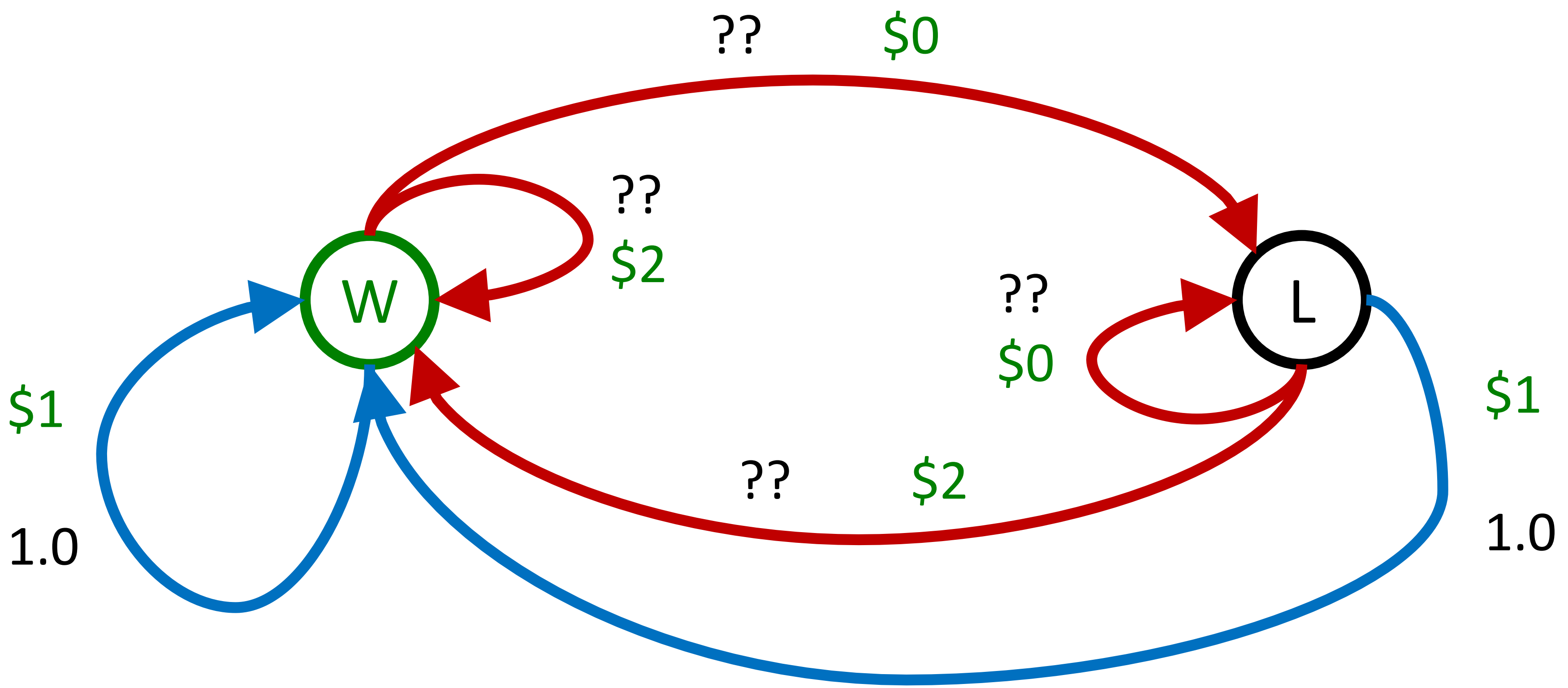


$2 $2 $0 $2 $2

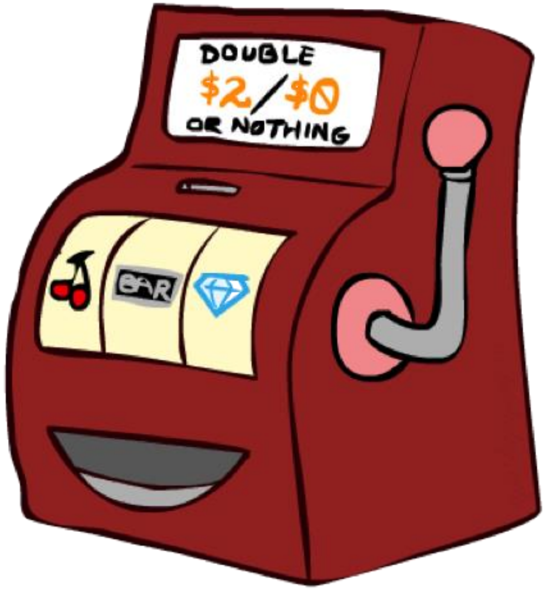
$2 $2 $0 $0 $0

Online Planning

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



Let’s Play!



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

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

What Just Happened?

- That wasn't planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn't solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP

Next Time: Reinforcement Learning!