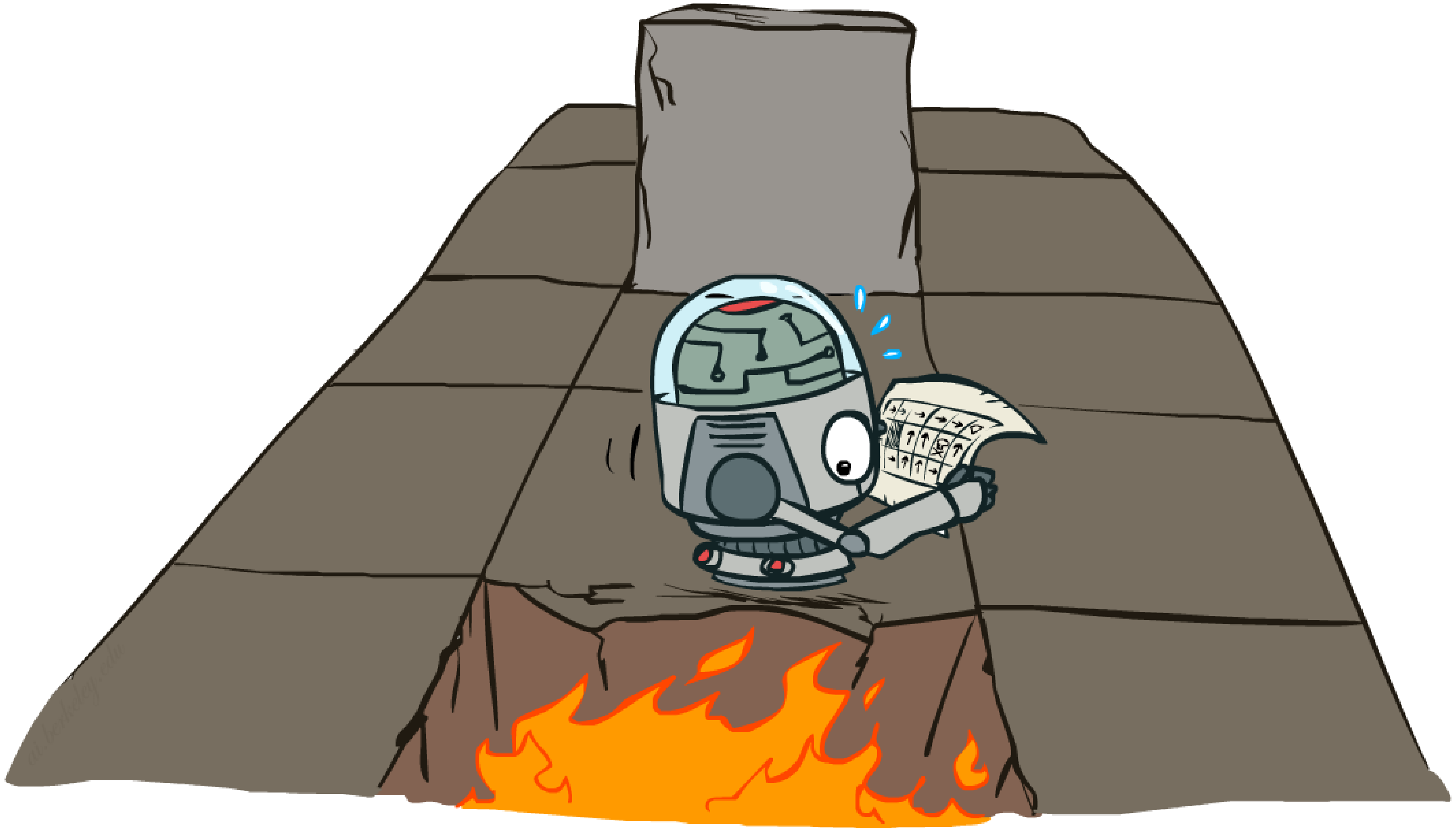
CS 188: Artificial Intelligence

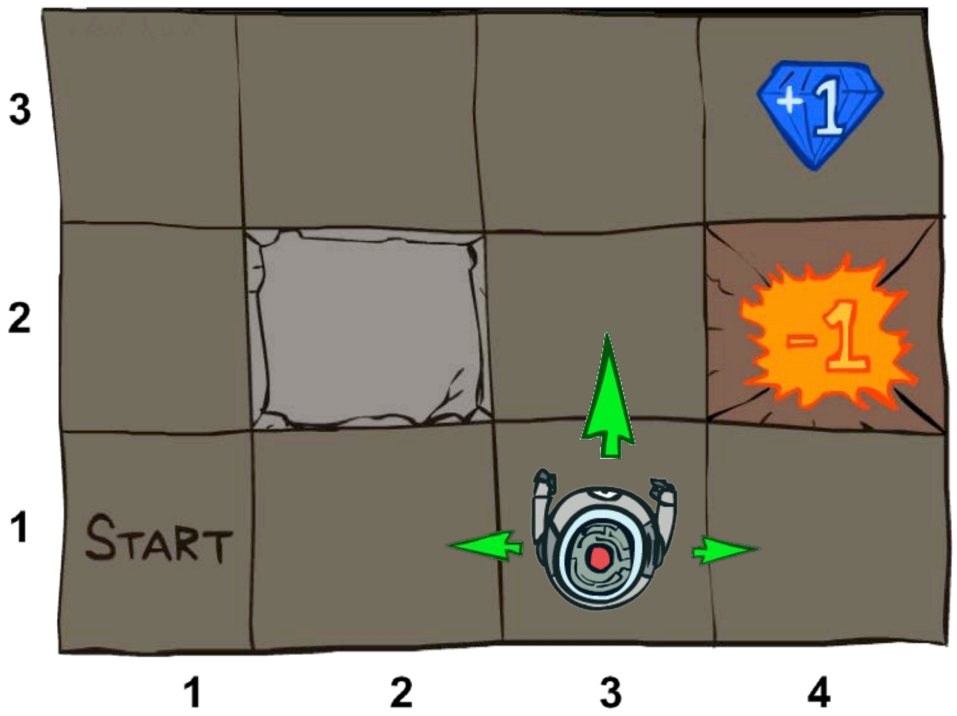
Markov Decision Processes II



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

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
  - 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 (discounted) rewards

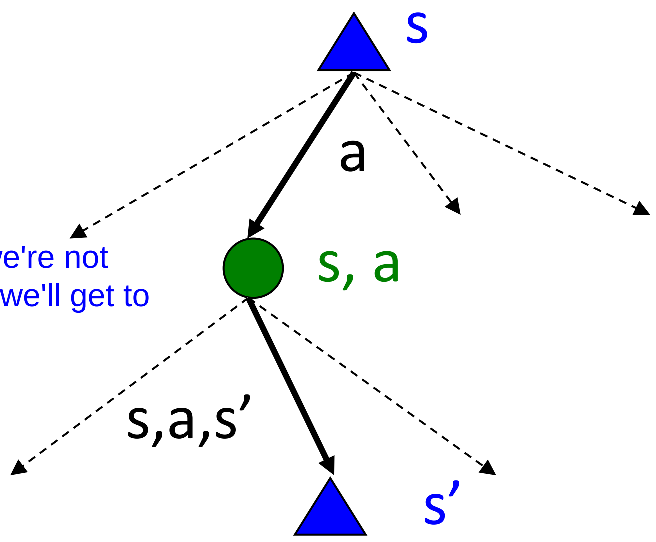


MDPs are non-deterministic search problems

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

MDPs are different from search in that there's a transition function



at this point we're not sure which s' we'll get to

- 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:

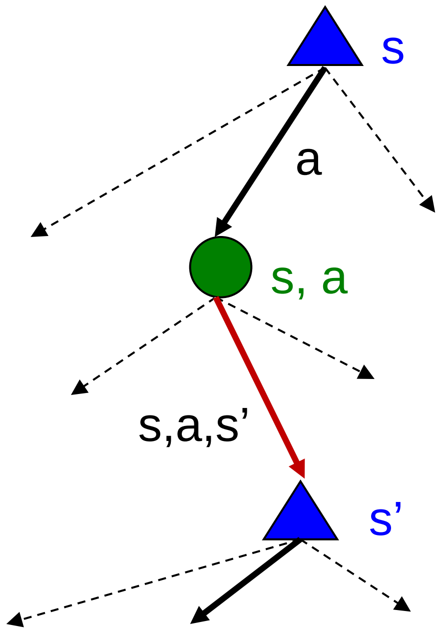
  $V^*(s)$ = expected utility starting in s and acting optimally

- 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



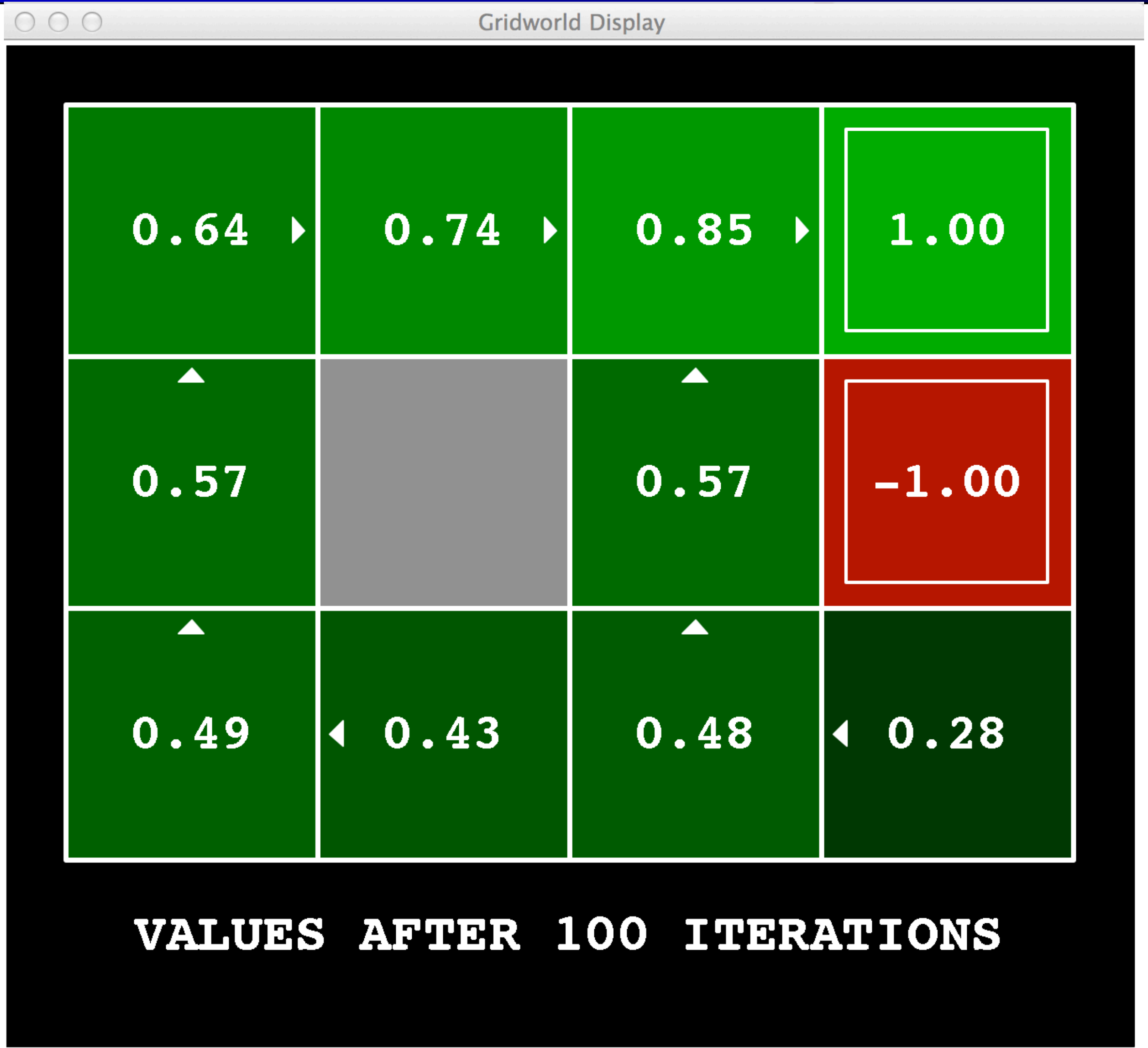
s is a *state*

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

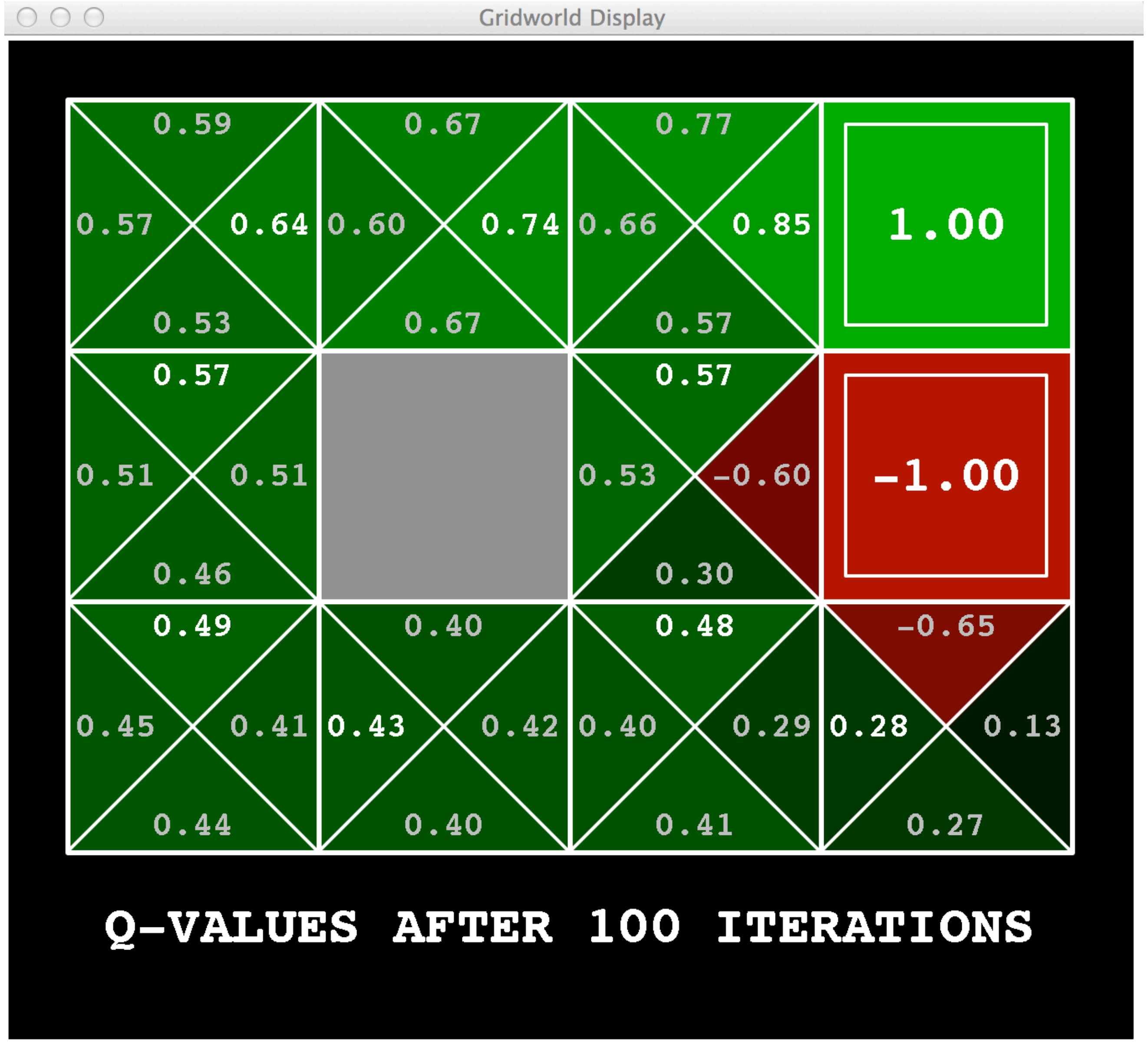
(s,a,s') is a *transition*

[Demo: gridworld values (L9D1)]

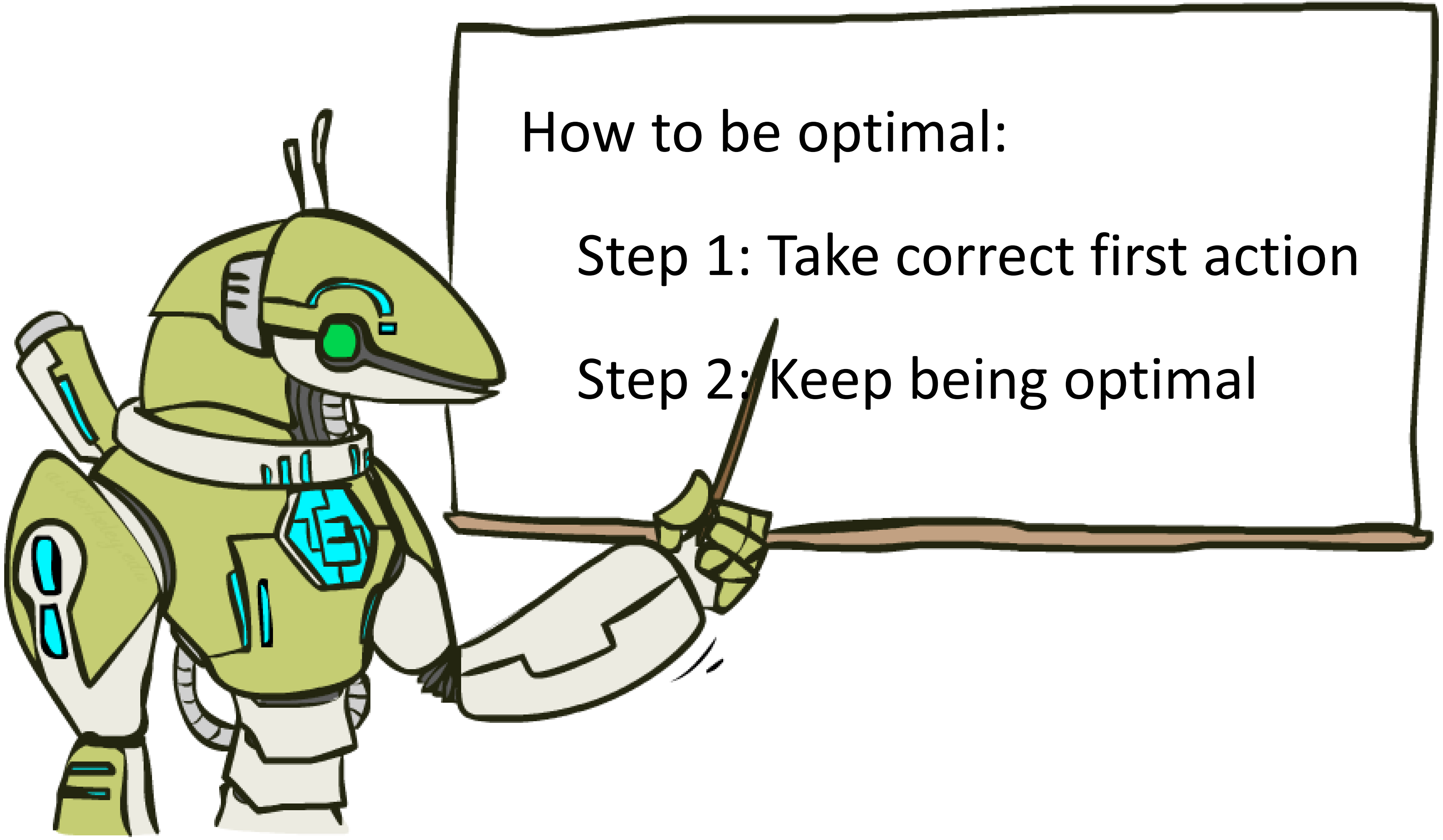
Gridworld Values V*



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

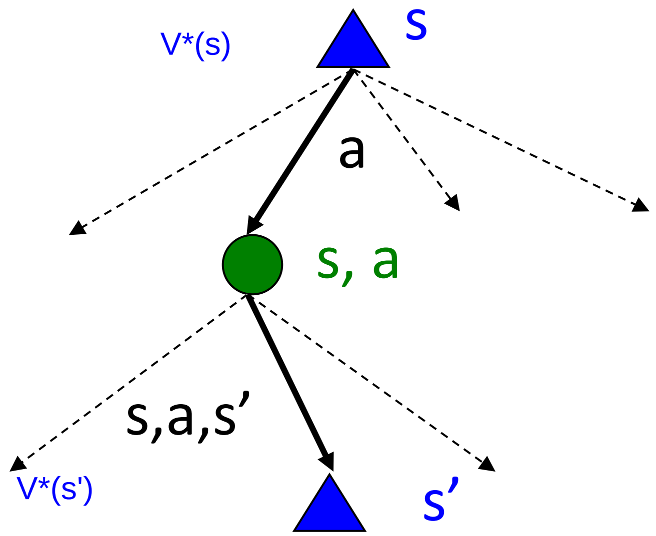
The Bellman Equations



Average over all the possible outcomes of being in state s and doing action a

The Bellman Equations

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



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

this is what I'll get from being in state s if I act optimally!

Q*(s,a)

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

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

immediate reward from making that transition

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

this characterizes optimal values but it's NOT an algorithm for computing them

Value Iteration

value iteration is an algorithm

- Bellman equations characterize the optimal values:

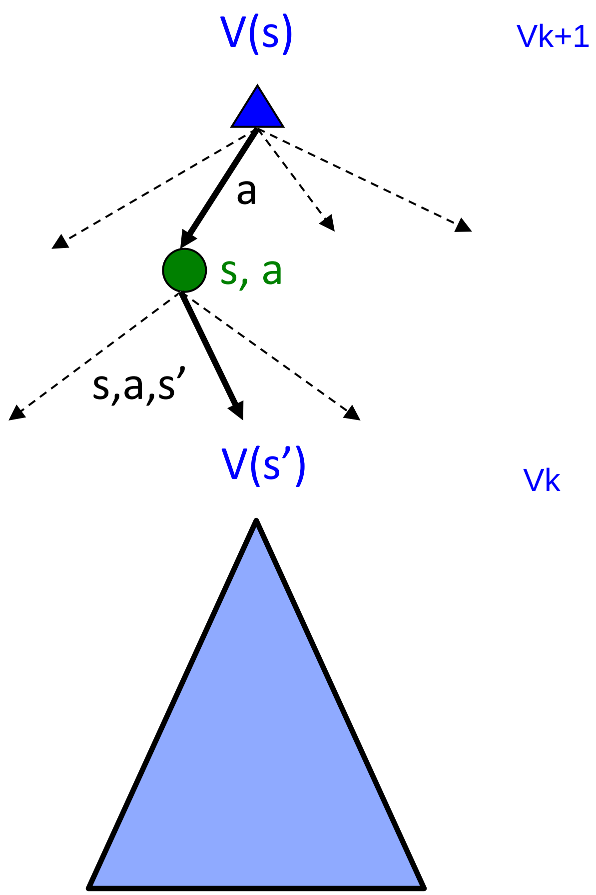
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

- Value iteration computes them:

if we have an approximation Vk for all the states (even if it's a bad approximation), we can get a new/ better approximation for all the states

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

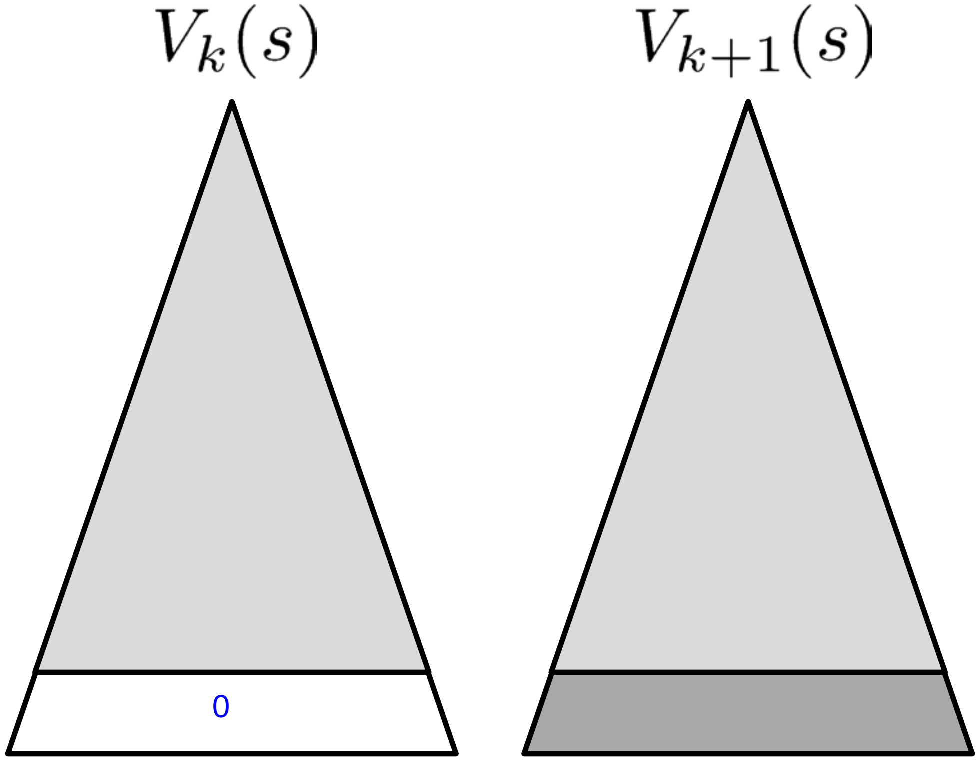
- Value iteration is just a fixed point solution method
  - … though the V_k vectors are also interpretable as time-limited values



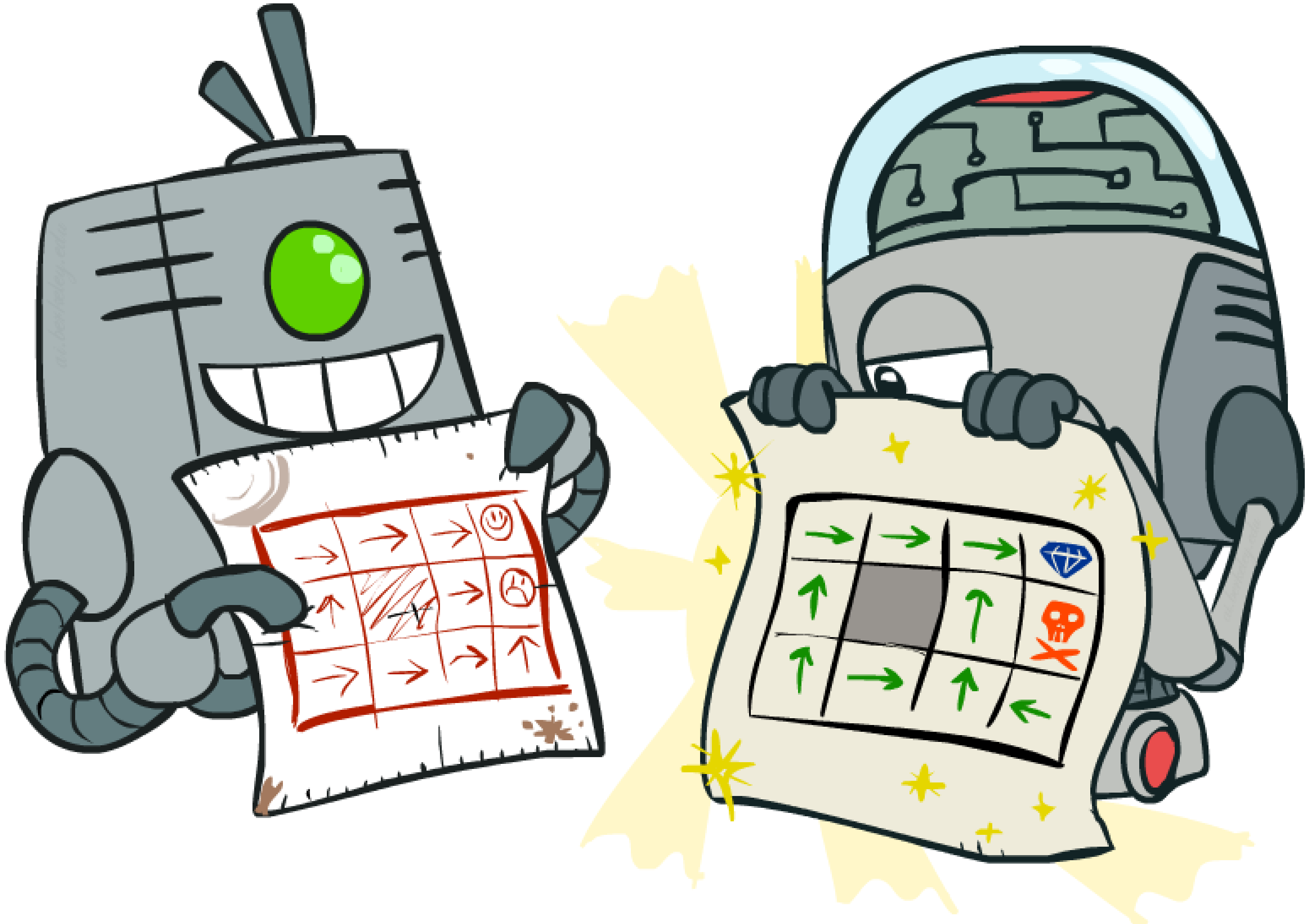
Vk, which is a vector of values for each state, is a sequence of approximations to the values.
Vk represents the expected utility for k steps, starting in a given state.

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



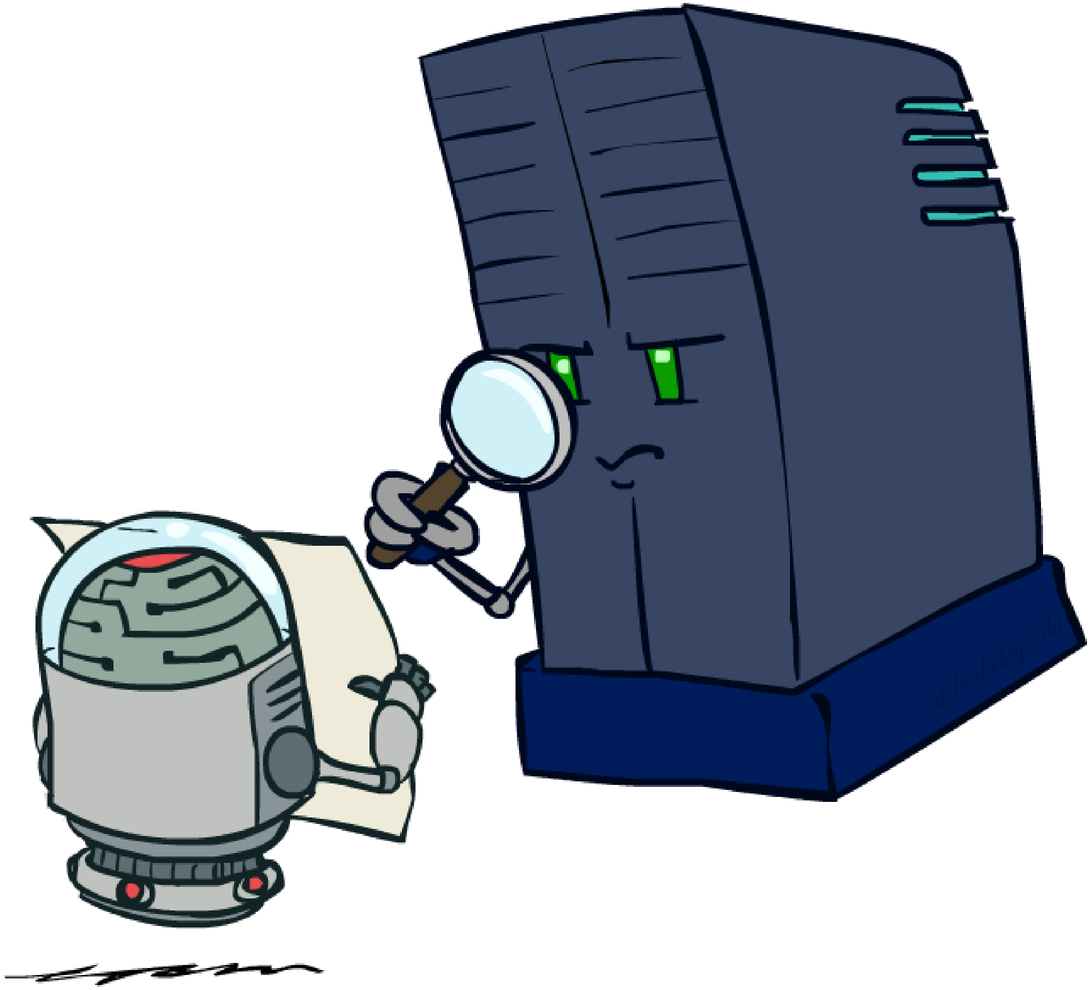
Policy Methods



Policy Evaluation

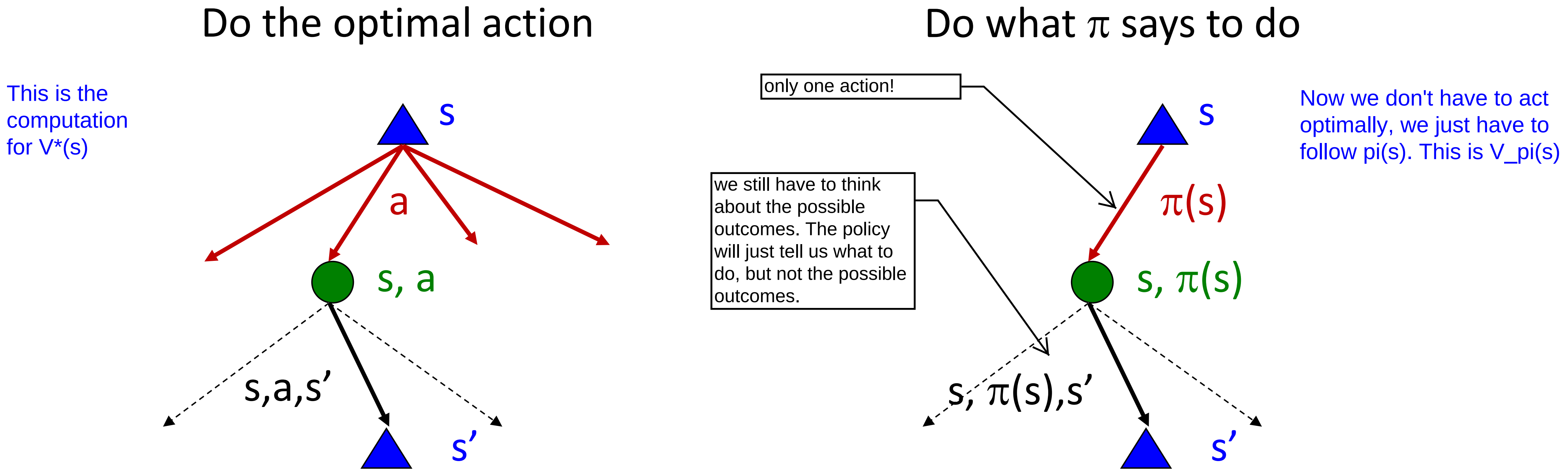
In Policy Evaluation, someone has given you a policy. That policy may be good, it may be bad. All we want to know and find out is how good that policy actually is. For each state, what will my score be if I do this policy?

Input: A policy
Output: A vector of values for each state. May not be optimal, unless the policy was optimal!



Computation is much easier if you have a fixed policy. All those "max's" go away. You don't have to find the best action because someone has already told you the best action! All you have to think about is the different possible outcomes.

Fixed Policies



- 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

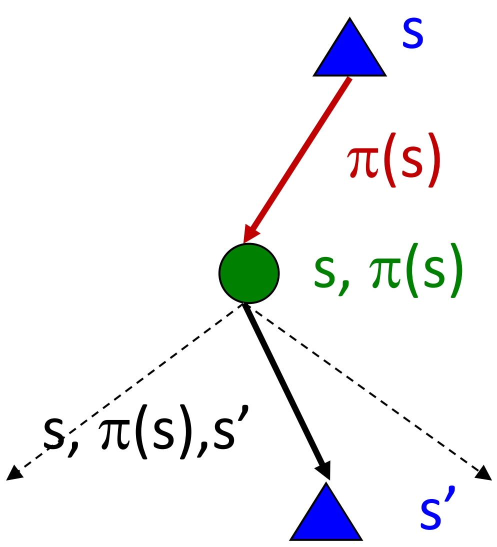
Utilities for a Fixed Policy

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

- 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):



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

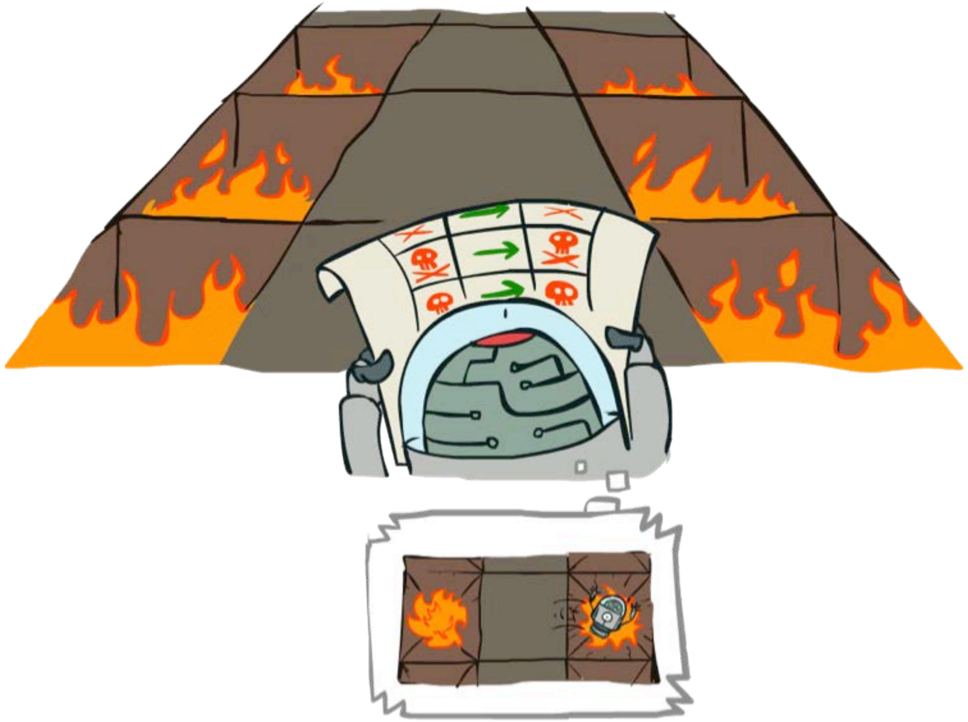
we don't need a max because we just follow pi(s)

a = pi(s)

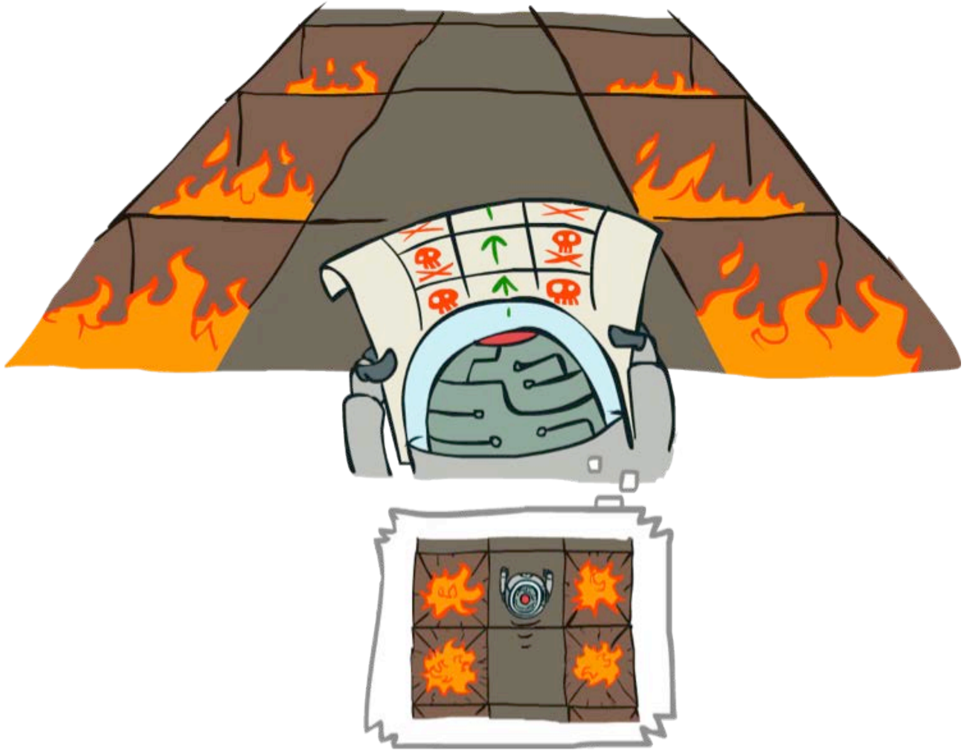
this is a linear system of equations

Example: Policy Evaluation

Always Go Right



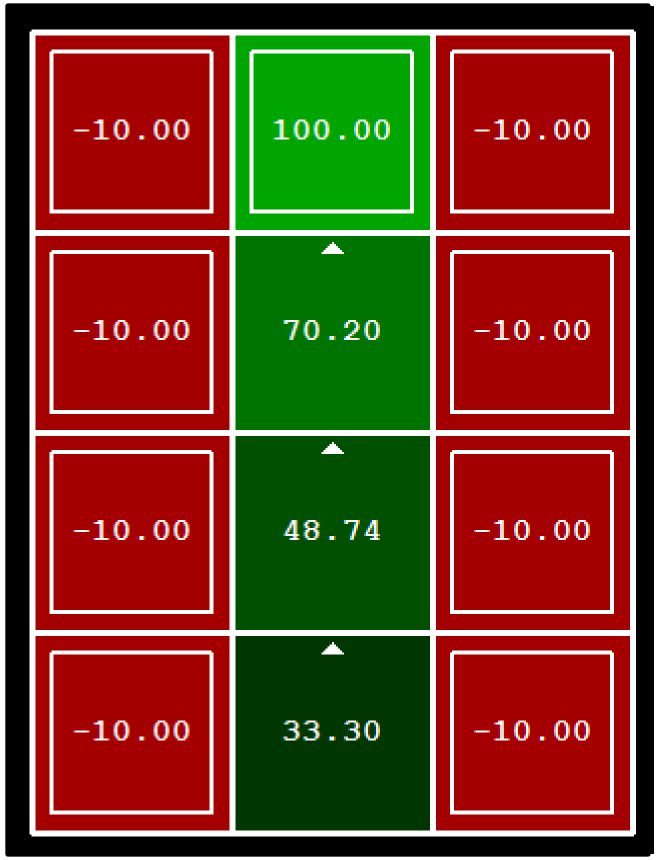
Always Go Forward



Example: Policy Evaluation

Always Go Right

Always Go Forward

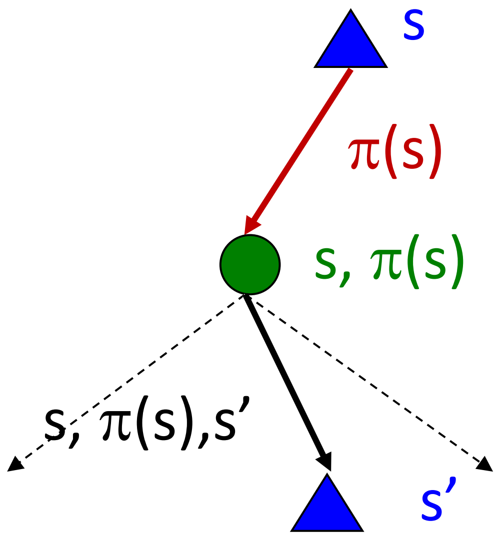


This is just Value Iteration
where you don't do a max

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)



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

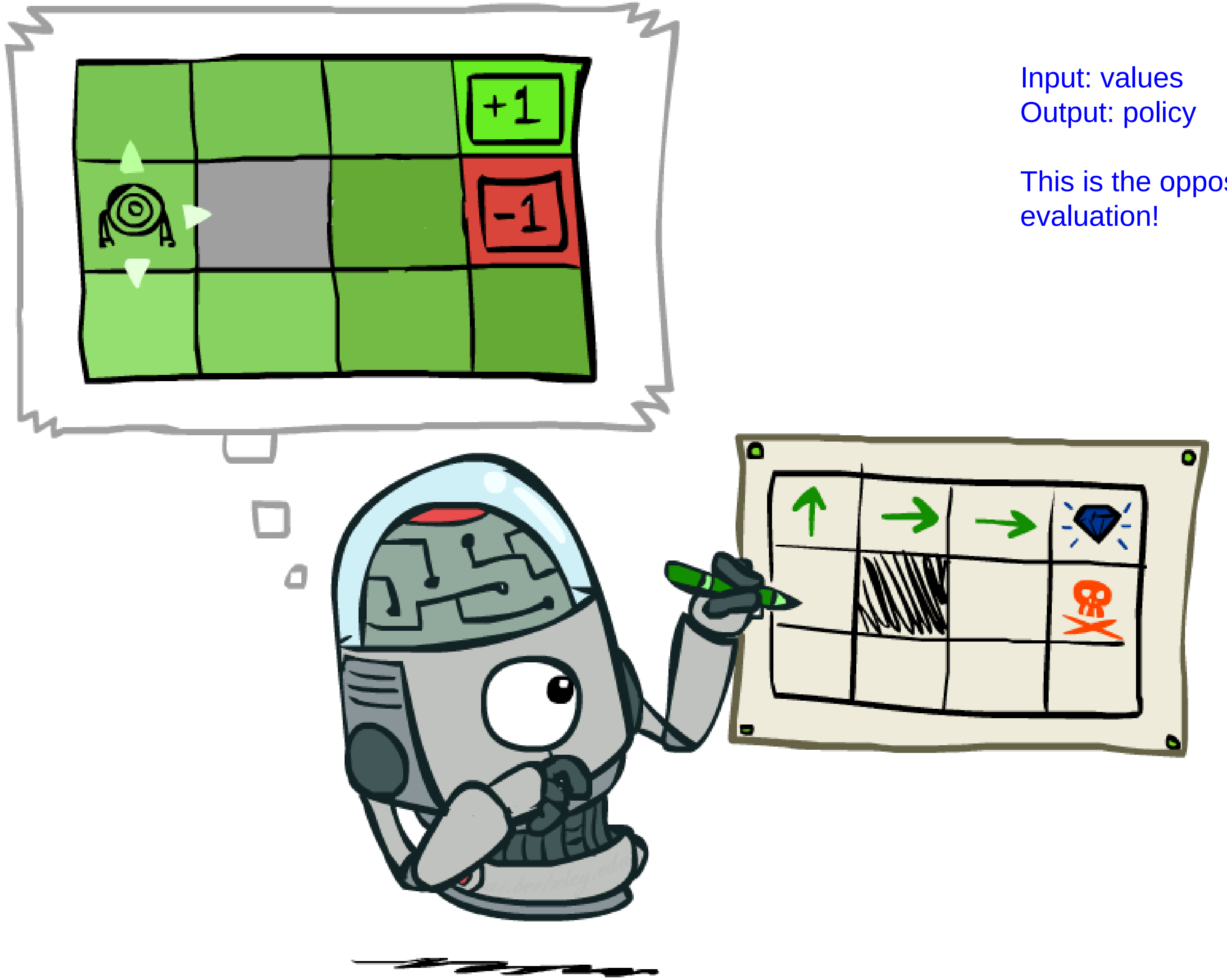
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

- Efficiency: O(S^2) per iteration

We have to do this for each state. And since the policy will land us in a new state. We also have to do this for each new state s'.

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

Policy Extraction



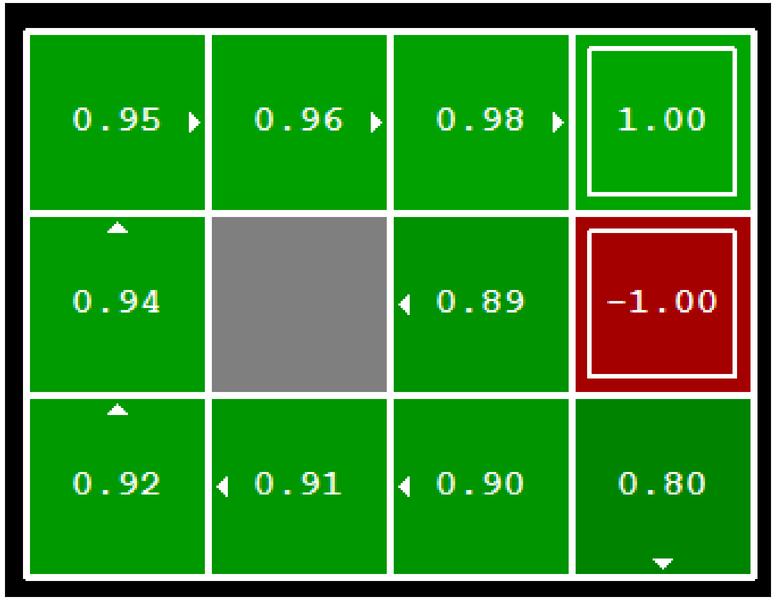
Input: values
Output: policy

This is the opposite of policy evaluation!

Computing Actions from Values

someone gave us optimal values

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



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

while the graph to the right shows the optimal policy, if we were at the 0.89 score, it's not obvious what action we need to take to go north? We might naively think we should go up, but we can see that the optimal action is to actually go left and "shimmy" our way over to the northern tile

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

we have to do a calculation over every a to find which action will give us the value we know to be true

$$\pi^*(s) = \arg\max_a \sum_{s'} \underbrace{T(s,a,s')[R(s,a,s') + \gamma V^*(s')]}_{\text{this is the calculation of a q-value}}$$

we pick that action that got us the maximum q-value

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

this is kind of annoying because we STILL have to do a one-step expectimax!

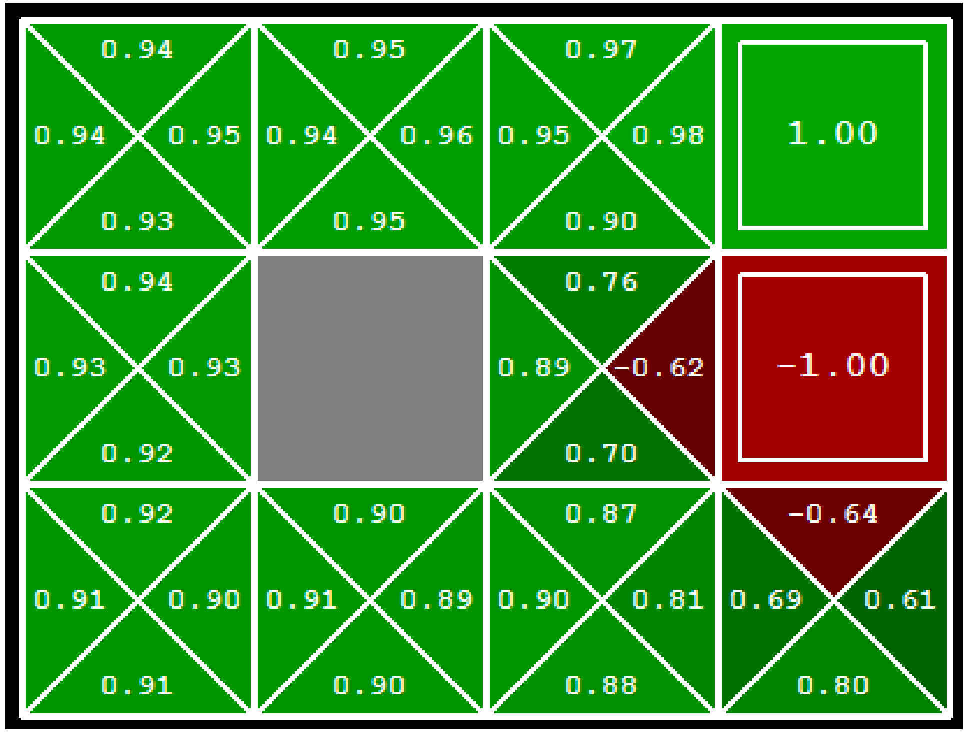
Computing Actions from Q-Values

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

  Q-values are really nice for policy extraction because it becomes really easy. It's much easier than if we the values V(s). The action is just whichever one gave you the highest q-value...So you just need to look at all q-values.

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

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



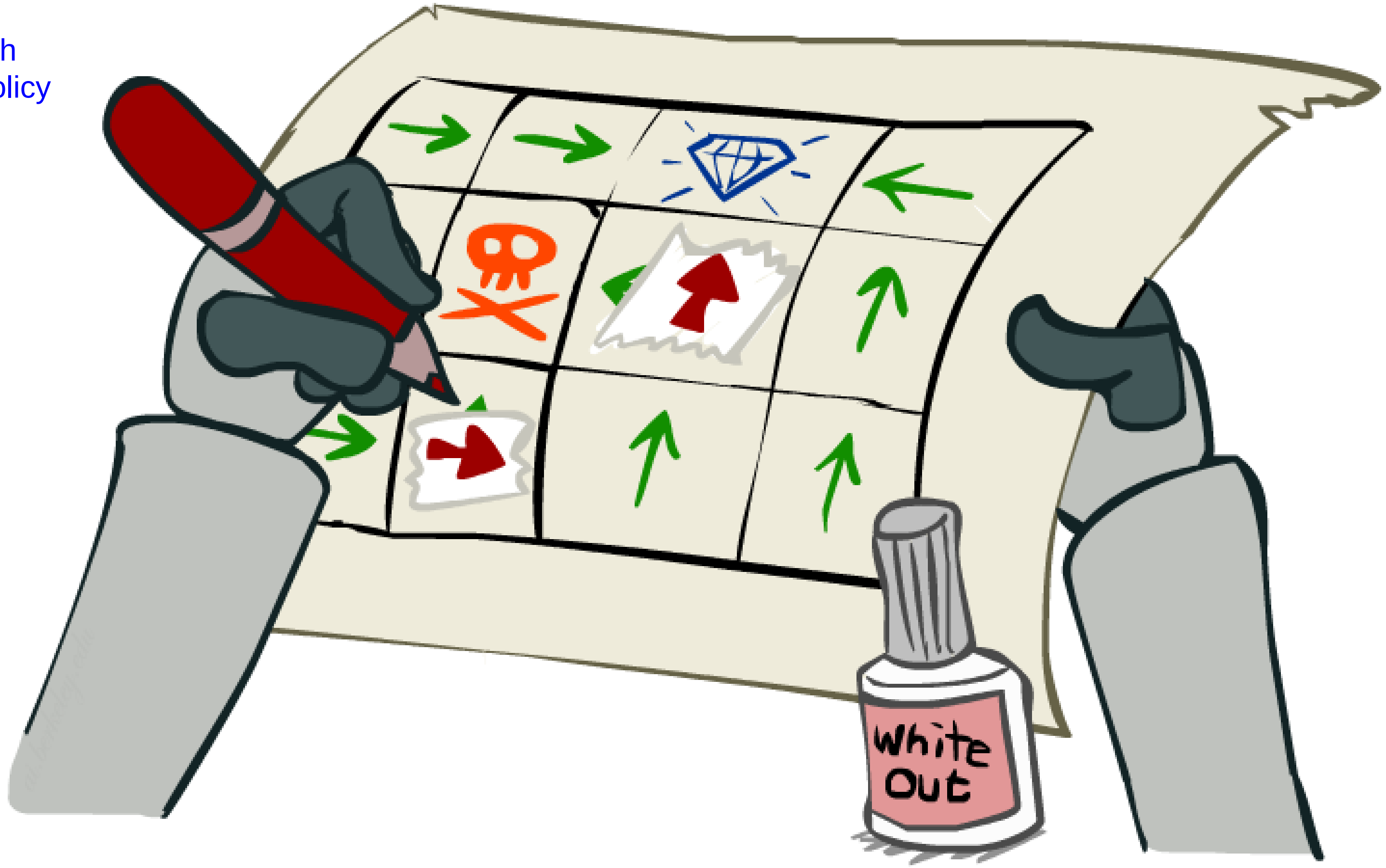
- Important lesson: actions are easier to select from q-values than values!

Policy Iteration

Once you have policy evaluation, which takes a policy and gives you values..

and you have policy extraction, which takes values and figures out what policy they imply...

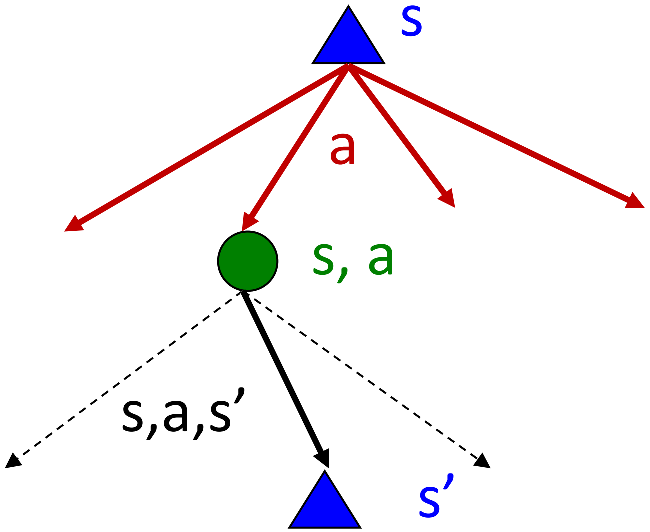
policy iteration is a simple algorithm where you just alternate these two!



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

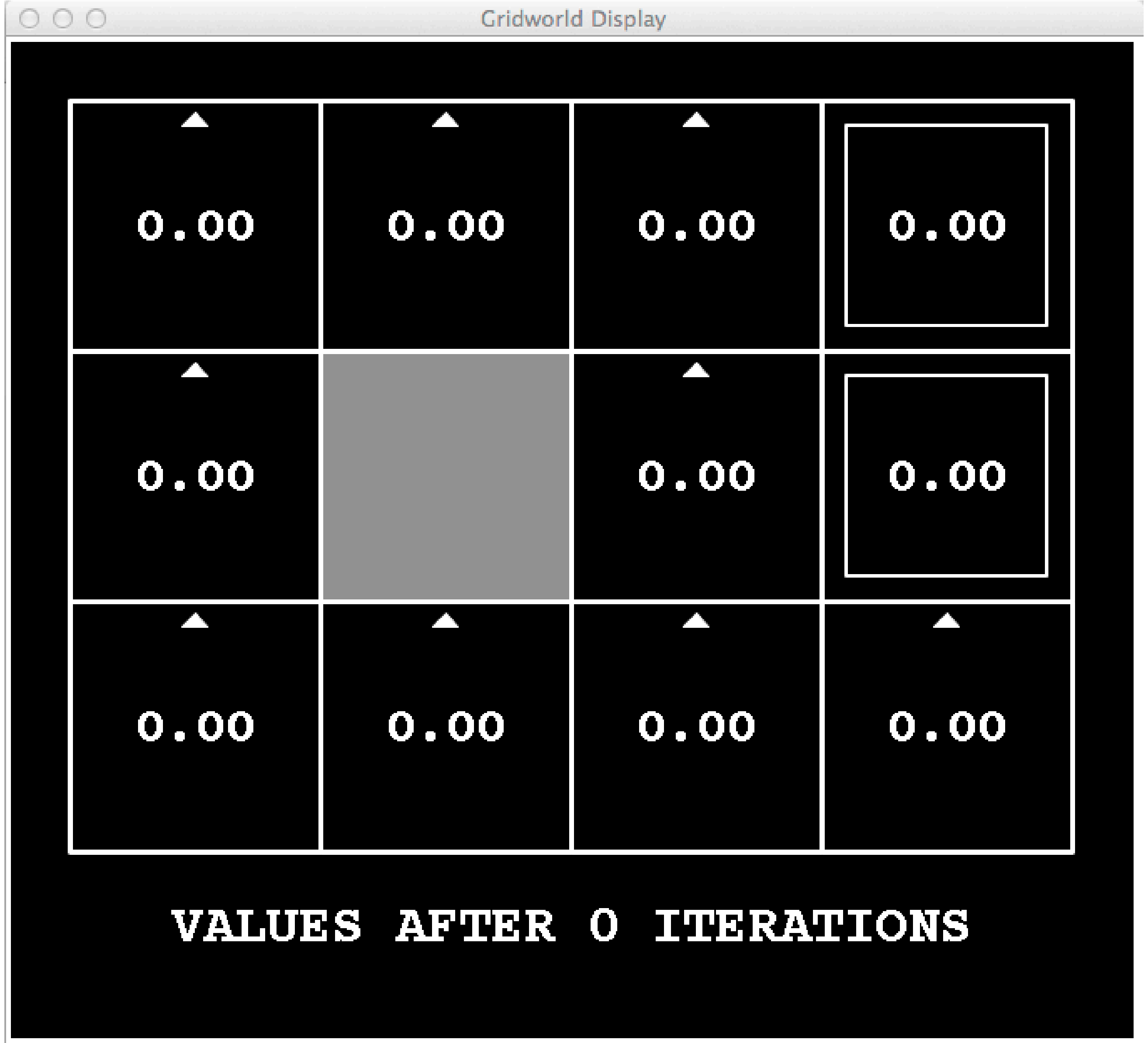
so all computation for the non-max's is wasted! We'll never use them

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

so all the computation after the policy converges is wasted

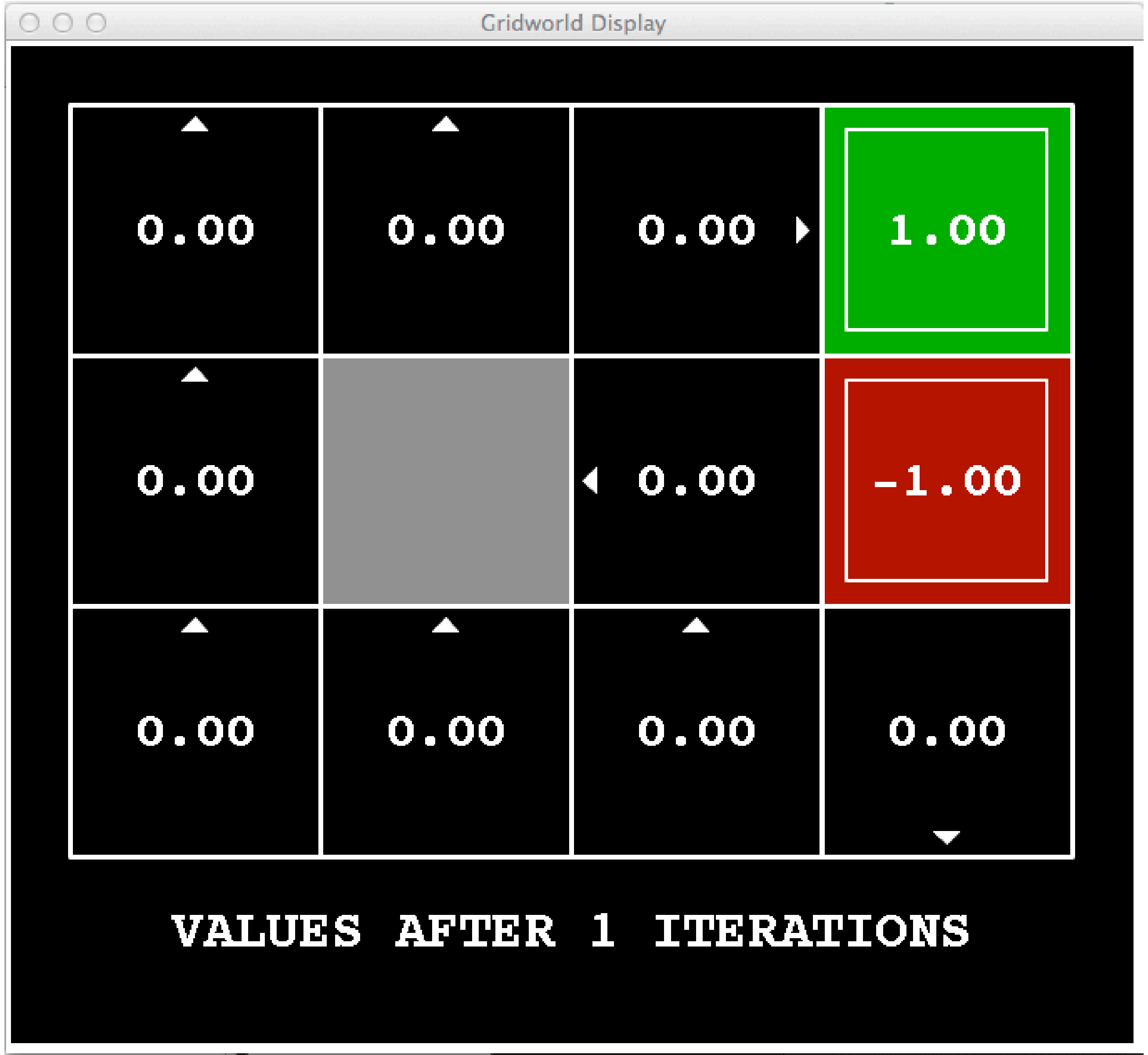
[Demo: value iteration (L9D2)]

k=0



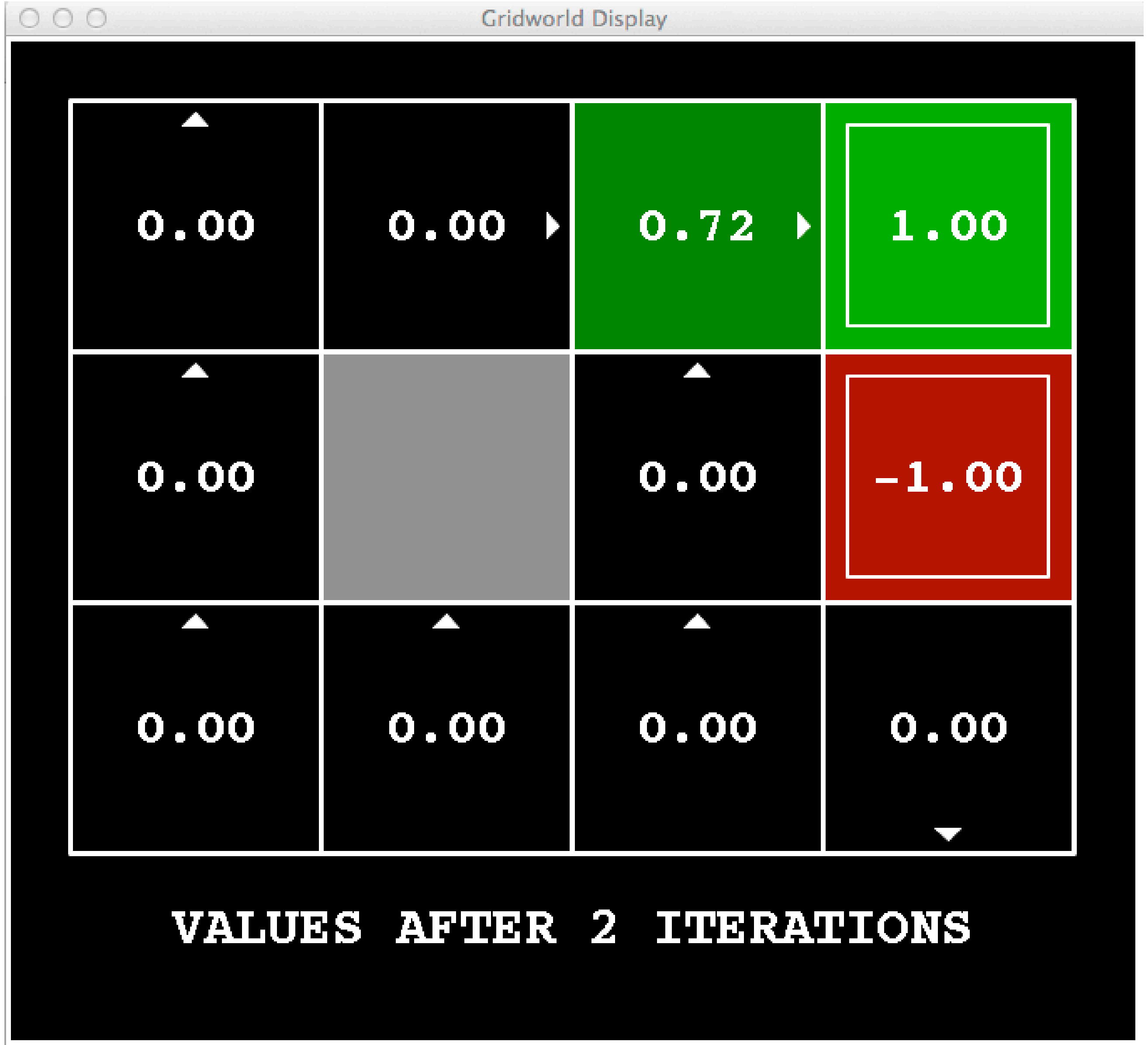
Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

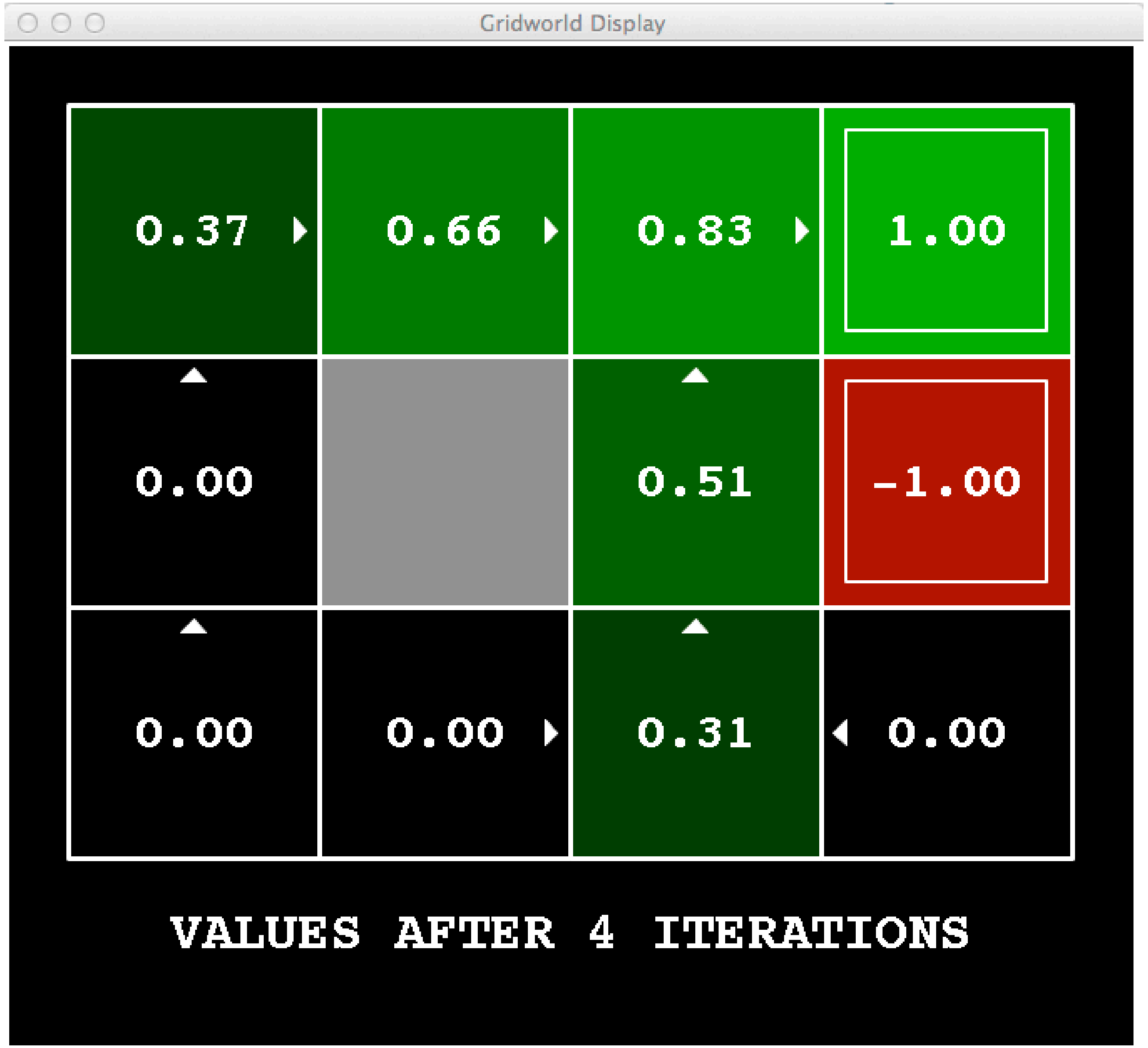


Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

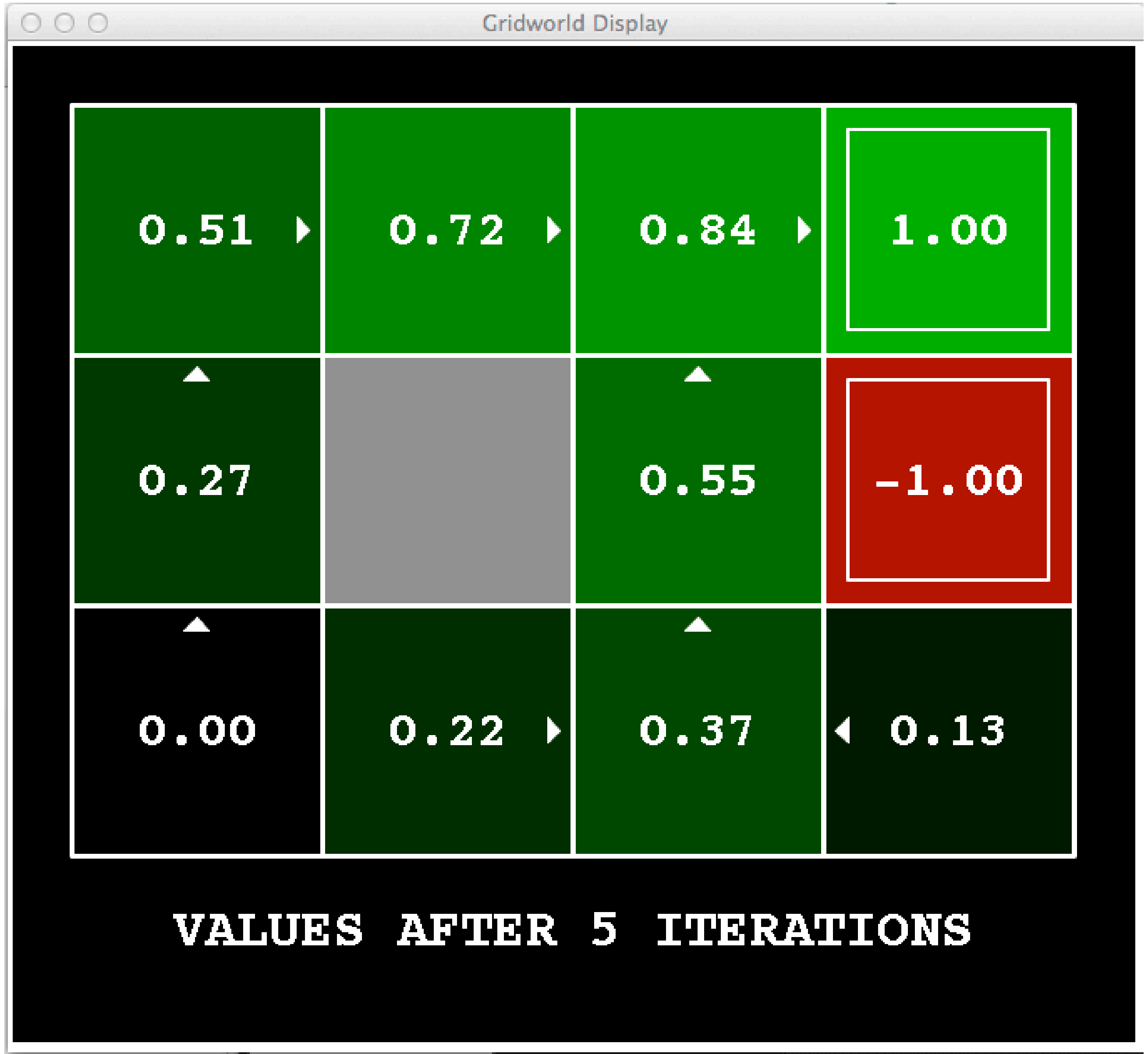
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



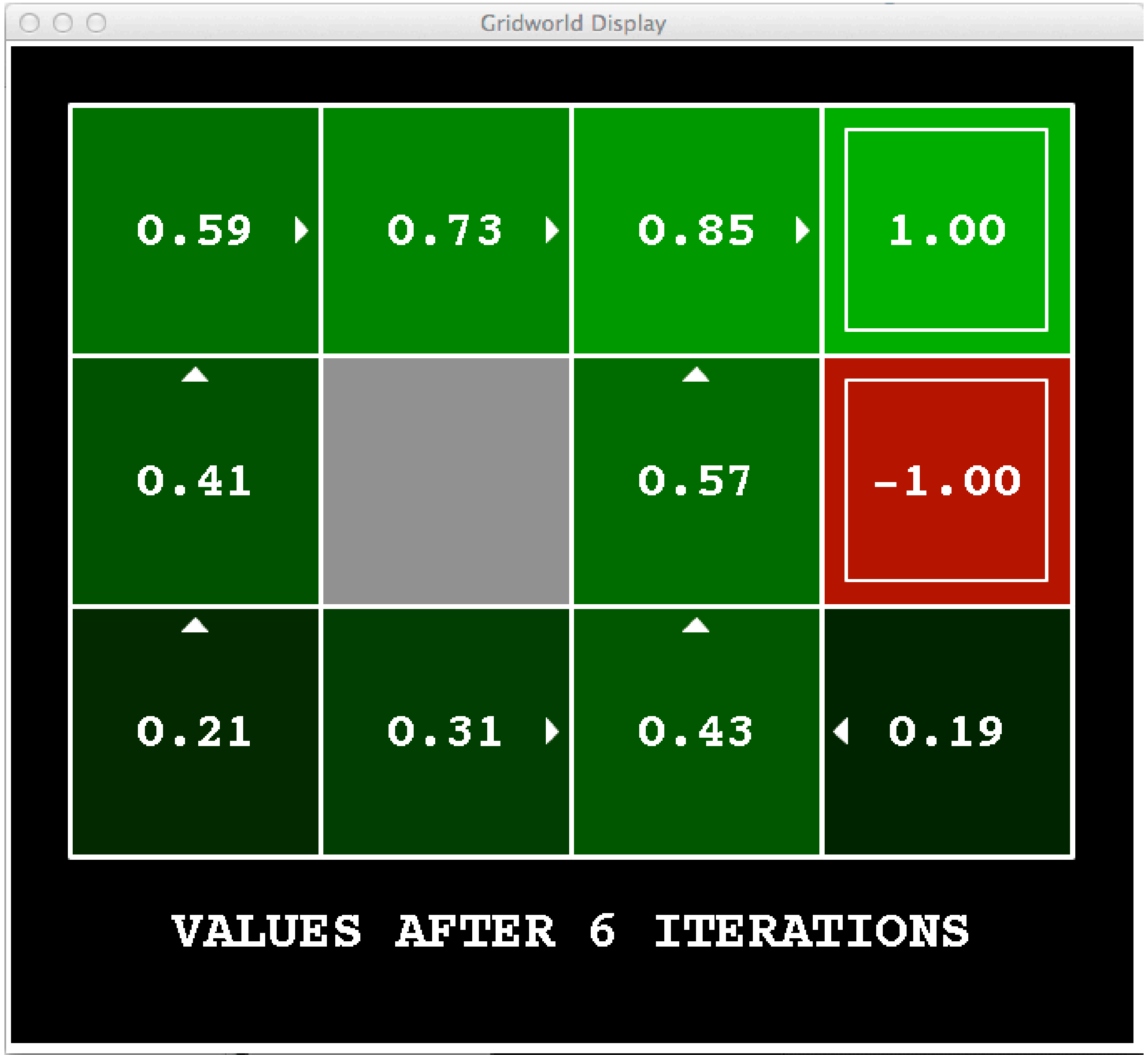
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



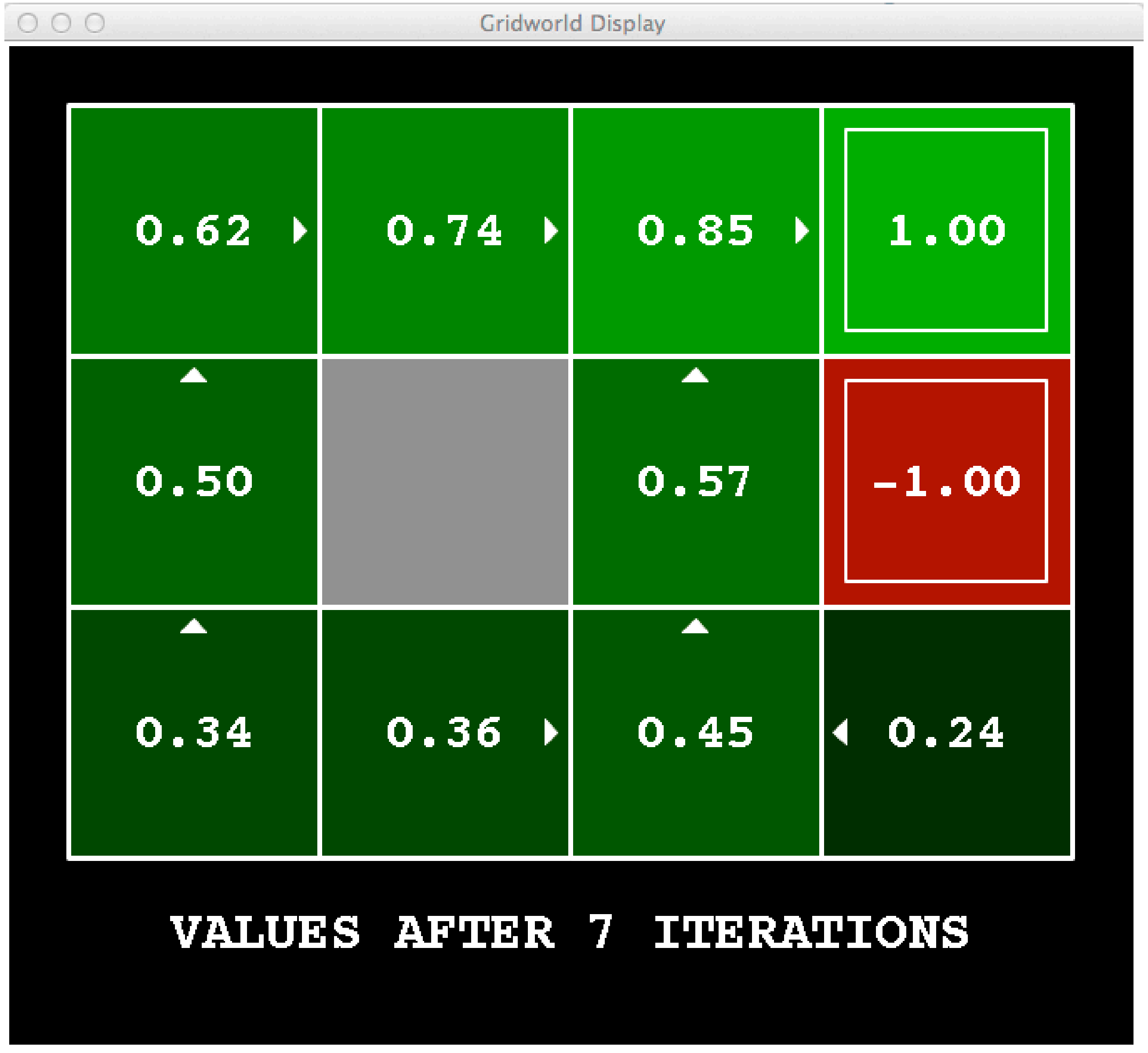
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



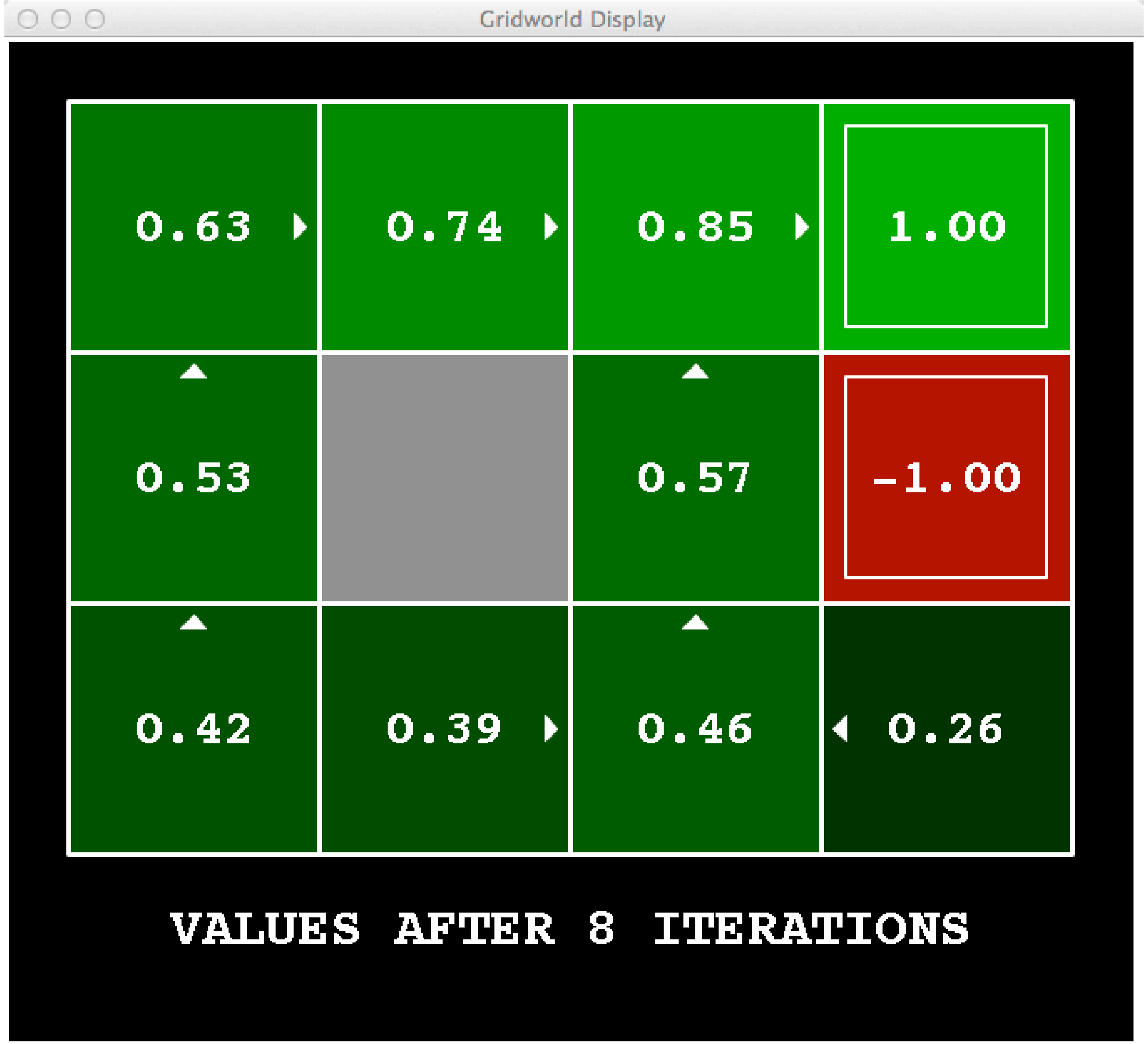
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



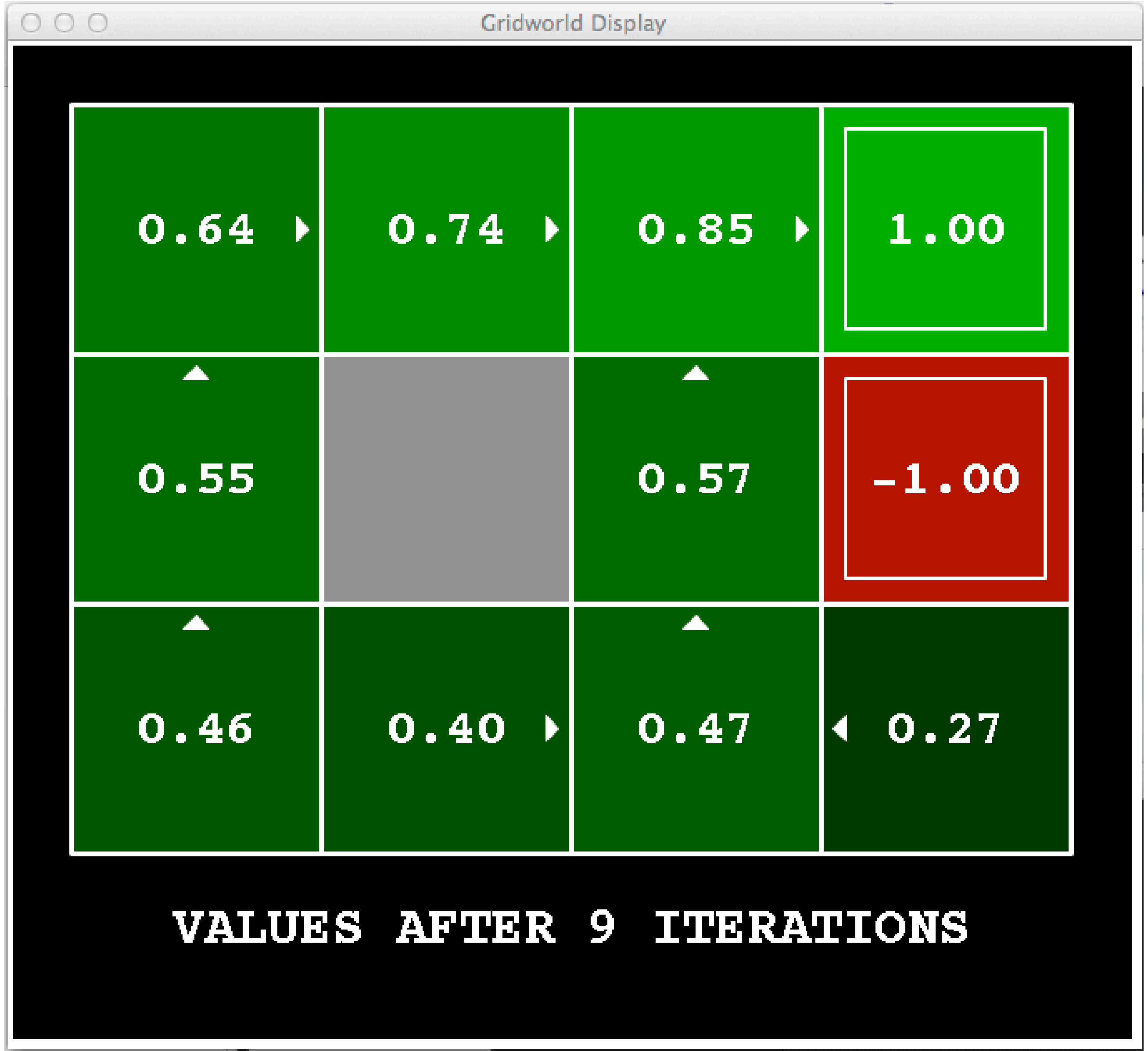
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



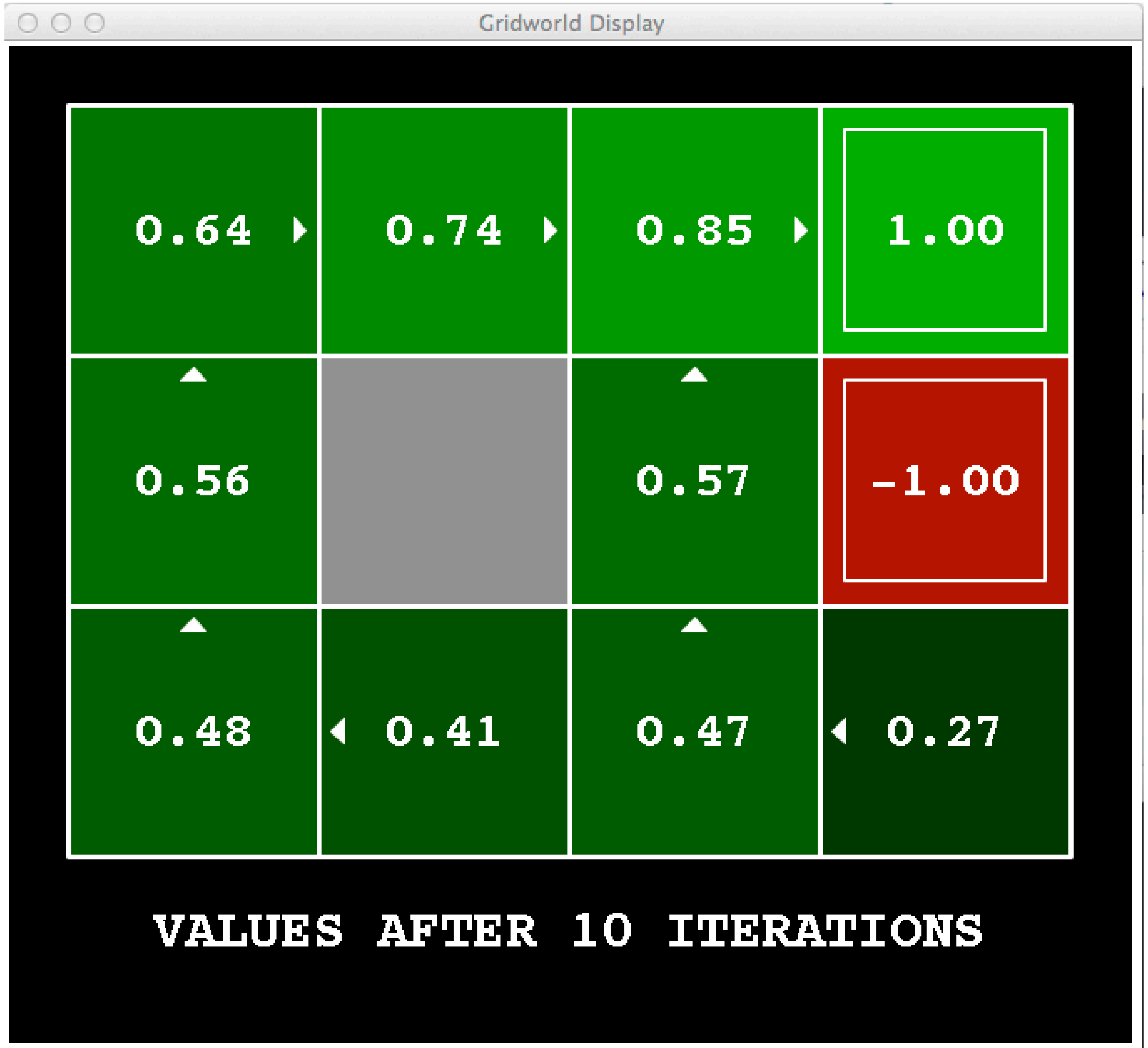
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



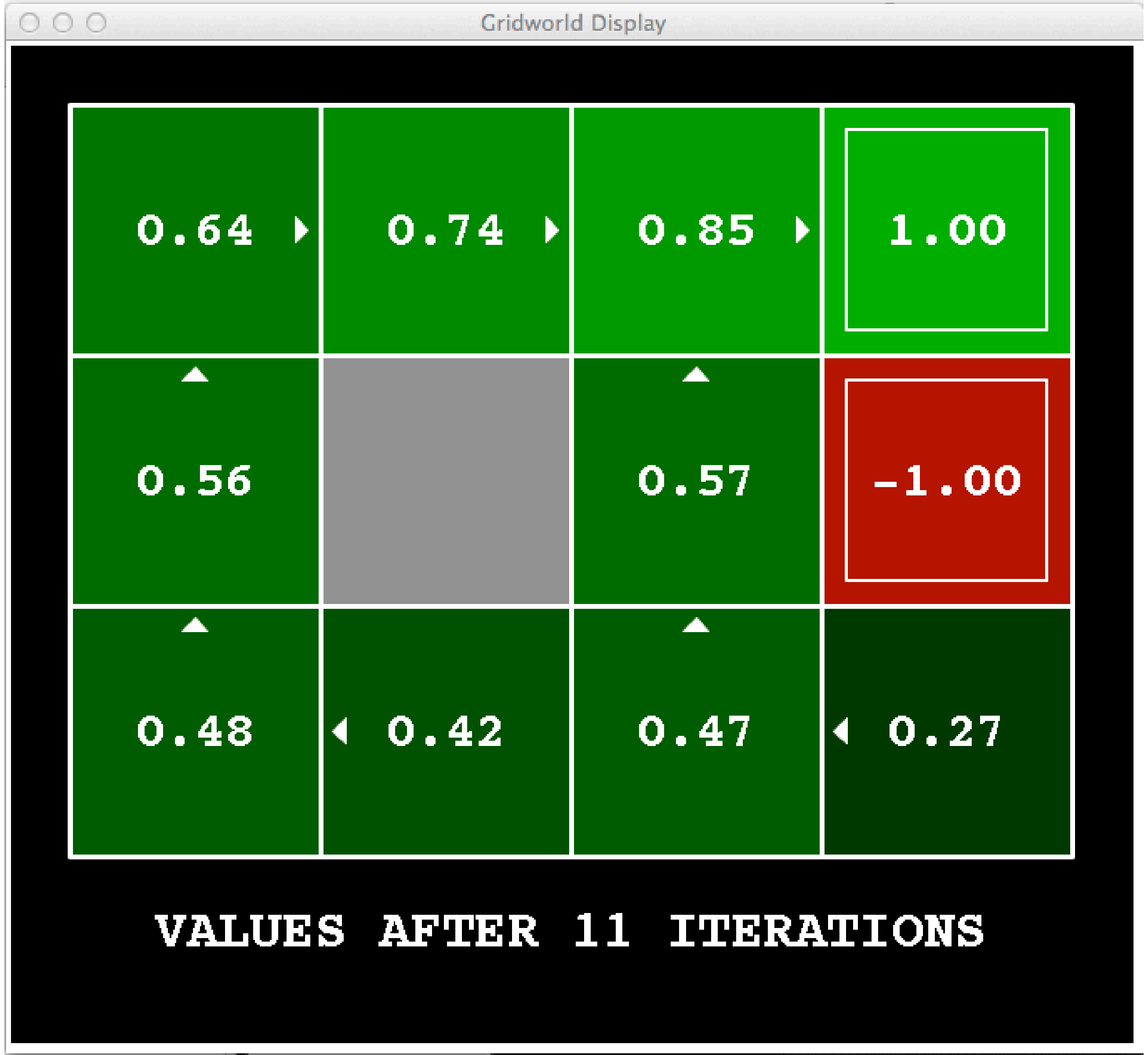
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



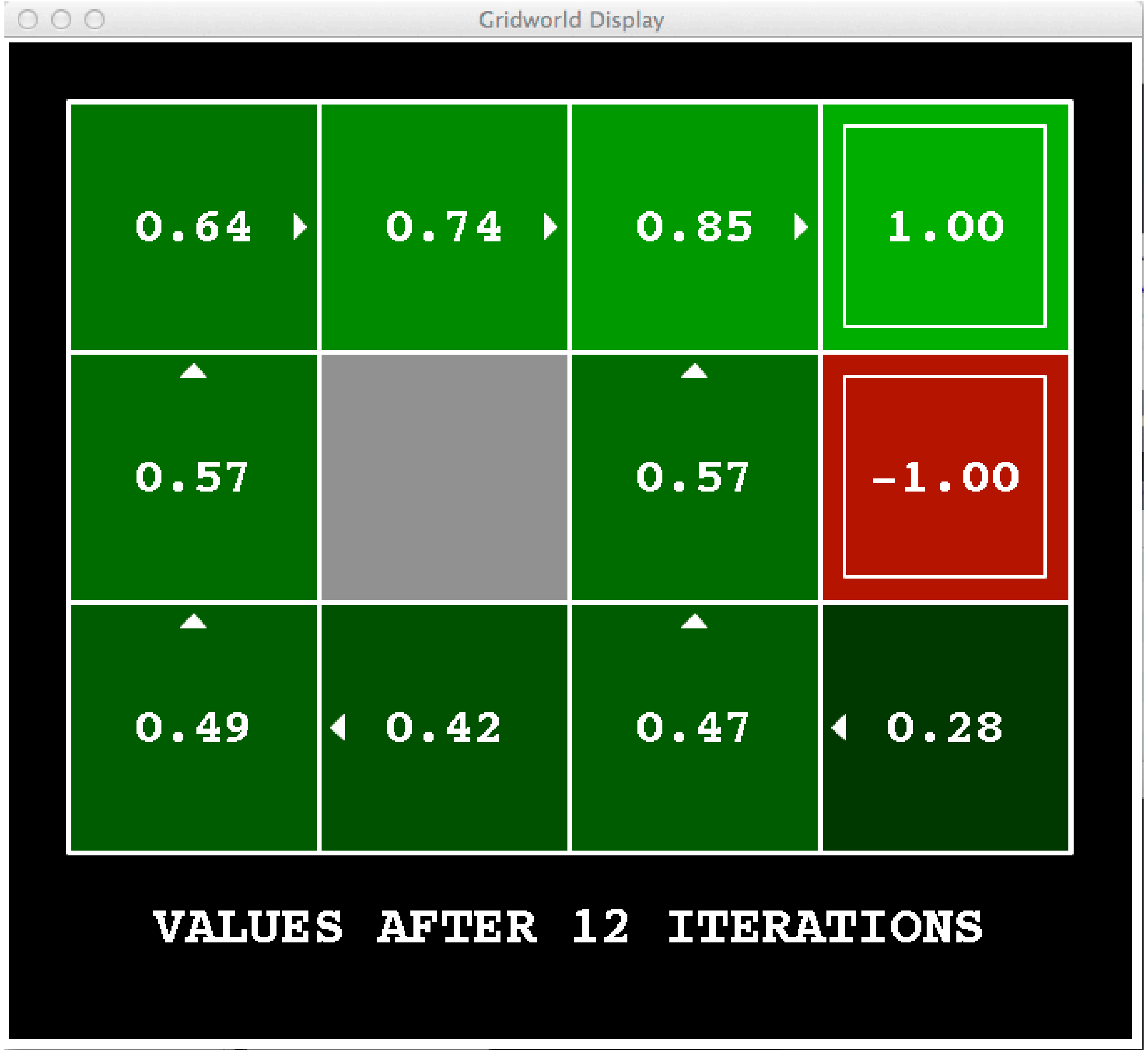
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



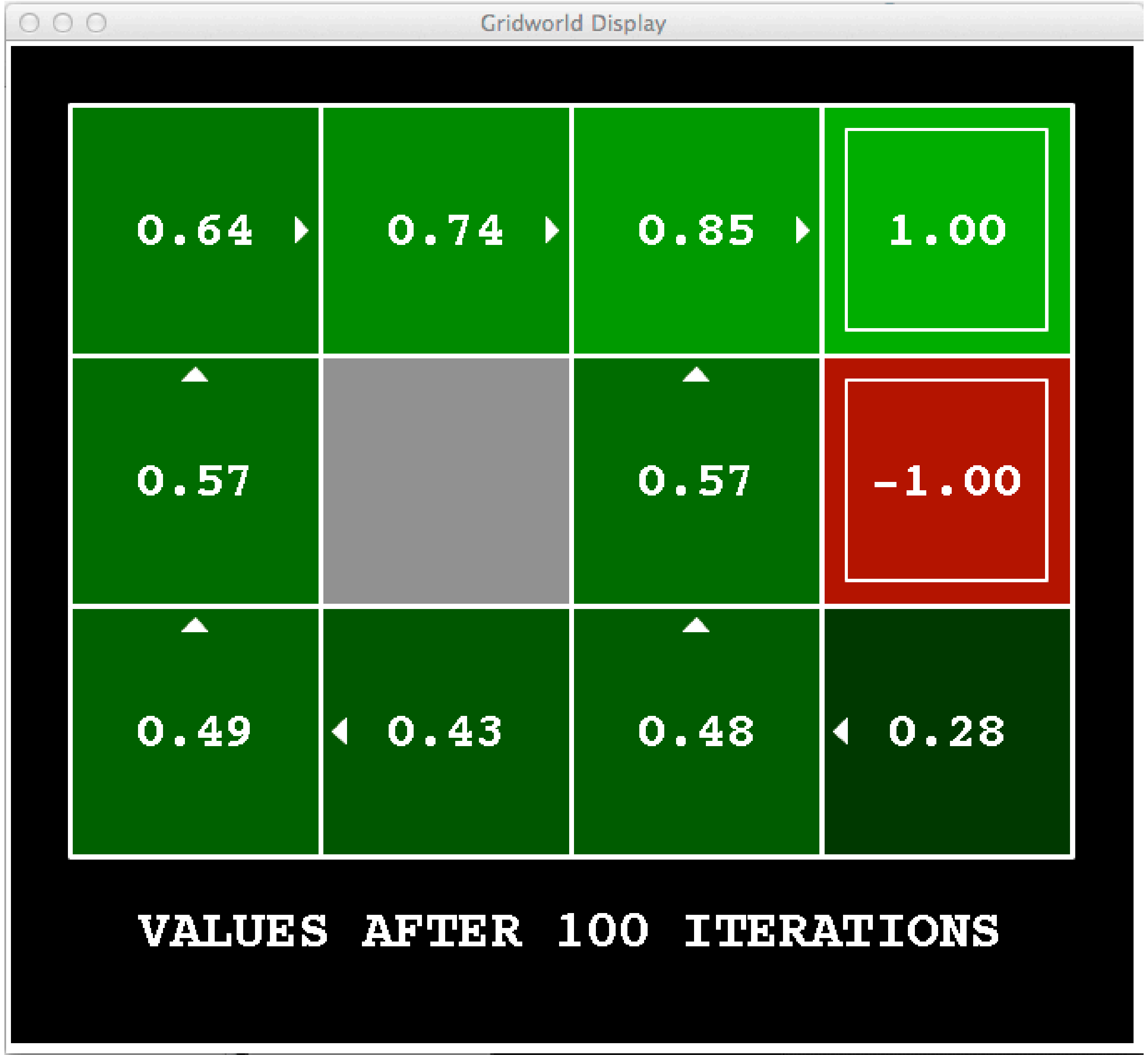
Noise = 0.2
Discount = 0.9
Living reward = 0

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

now that you have values for a policy (not the optimal policy, but some random policy) you will improve this policy! You'll do a one-step lookahead improvement round

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

Policy Iteration

presumably a bad policy

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

once it converges we know all the values of the states for policy pi

you do this a lot, until convergence, say 100x

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

"new"/updated values

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

this is the step that does all the work

this value came from the policy we just evaluated!

you do update the policy once, say 1x, for every iteration

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

I want my new policy pi_i+1. So now we take an argmax, we consider all actions, then average all the outcomes.

Both of these take an MDP and then compute for each state, the optimal value.

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: you usually keep the policy fixed, and you do a bunch of tracking of value changes under that policy, then every once in a while you let the policy consider other actions
  - 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

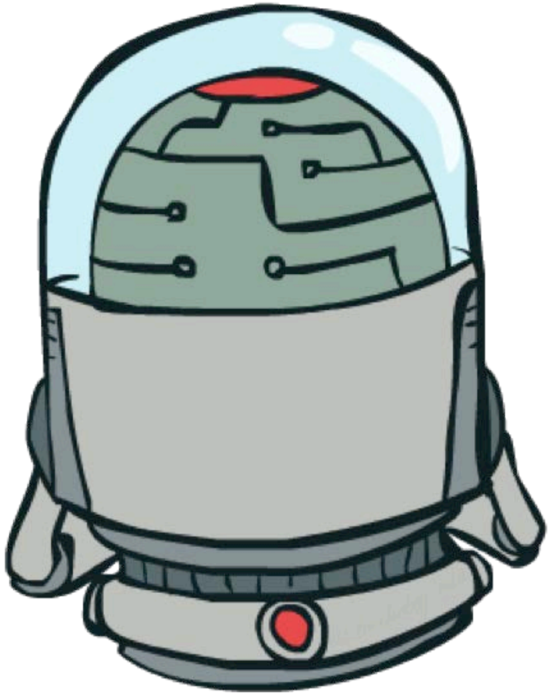
- So you want to….
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- 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

always gives
$1

gives you
either $2 or
$0



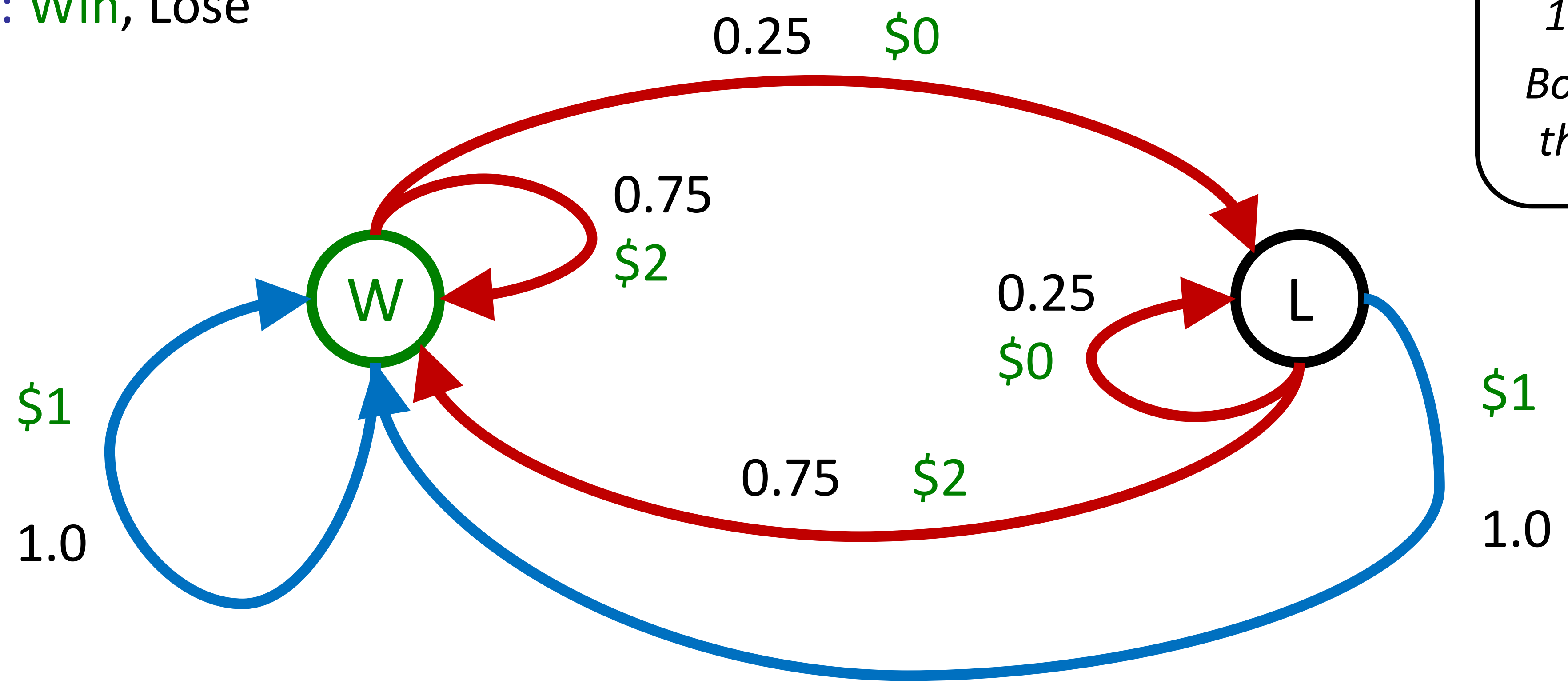
Double-Bandit MDP

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

No discount

100 time steps

Both states have the same value



this graph is an over complication of the MDP, but we need to have two states to formulate it correctly

there's really only one state, but we split it into two states b/c in the MDP formalism, the reward depends on whether you win or lose, so s' needs to be different for whether you Win or Lose

Policy 1: Play blue always. Expected utility in 100 time steps = $100

Policy 2: Play red always. Expected utility in 100 time steps = $150.

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!

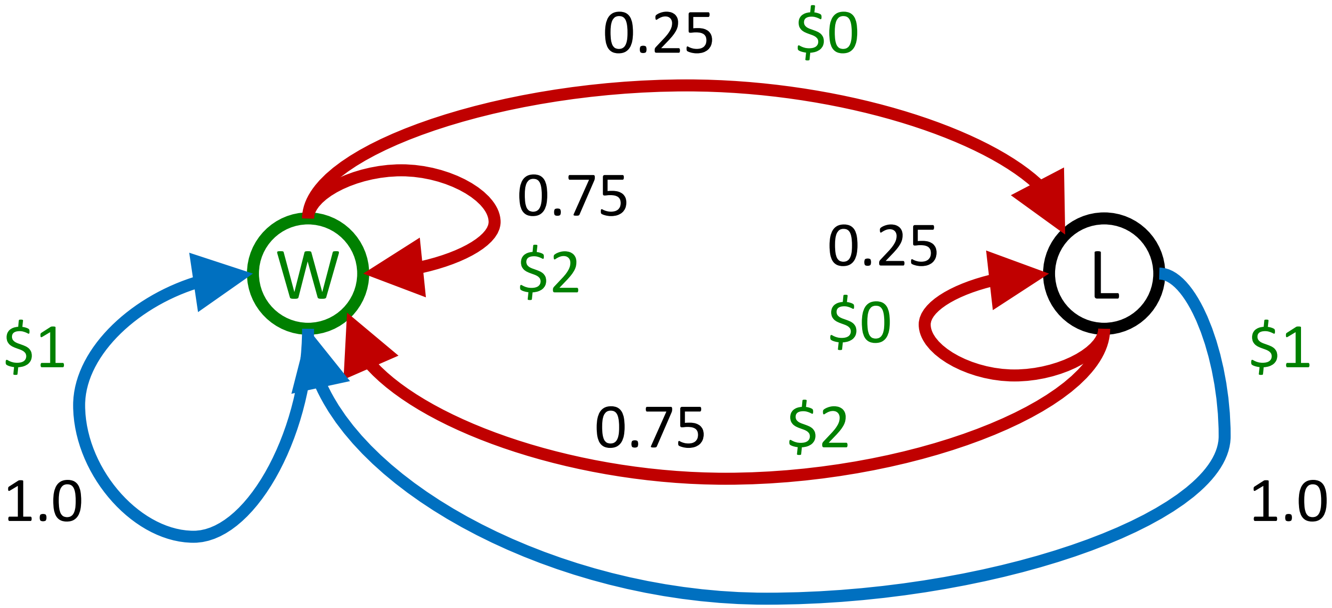
We determined the policy to Play Blue offline. We thought about possible policies, then determined which one we wanted to use. Then only after we figured out the policy will we play.

No discount

100 time steps

Both states have the same value

	Value
Play Red	150
Play Blue	100



Now let's actually play the game.

Let’s Play!

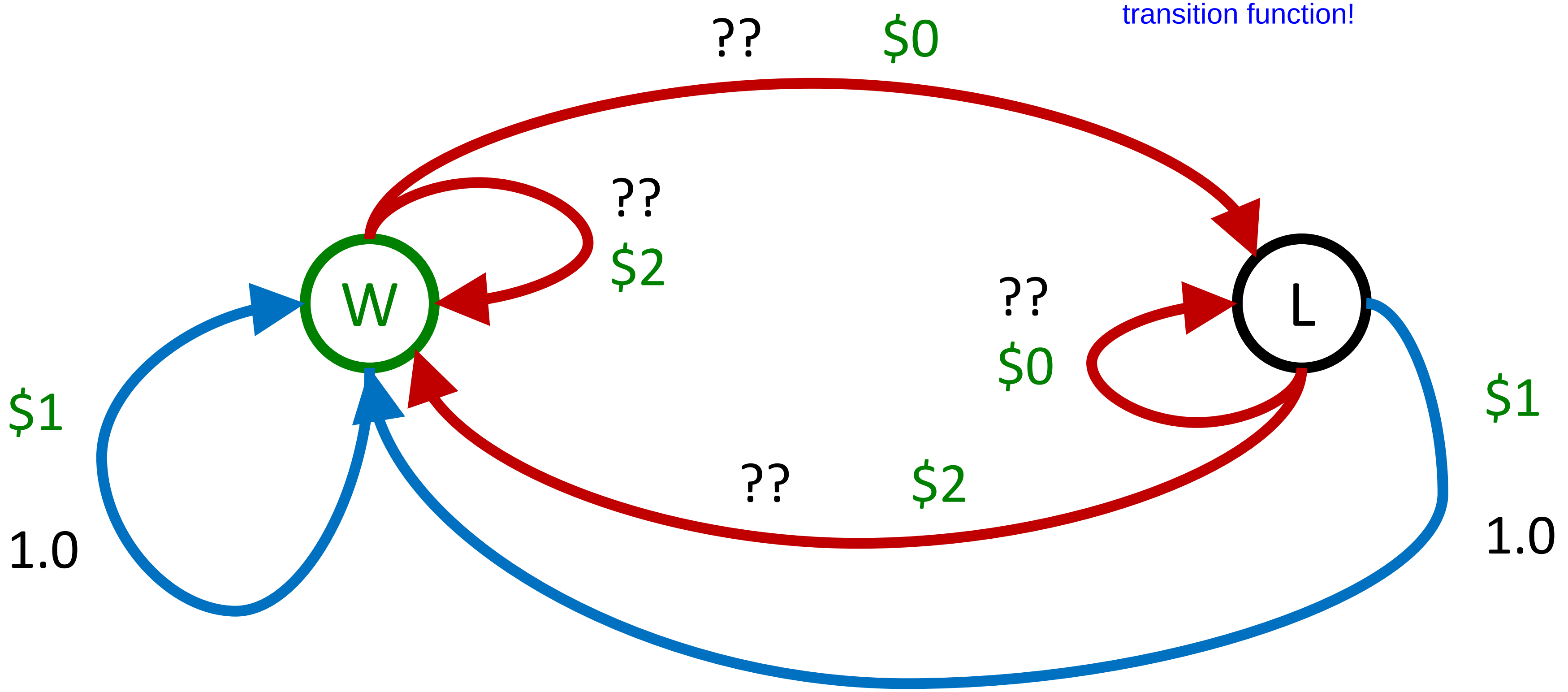
$2 $2 $0 $2 $2

$2 $2 $0 $0 $0

Online Planning

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

Same MDP in structure, but now we don't know the probability that we'll get $2 from pulling Red. We now CANNOT find the value of the Red policy ahead of time because we don't have the probabilities/ transition function!



Let's Play!

$0 $0 $0 $2 $0

$2 $0 $0 $0 $0

We didn't take the MDP and solve it because we didn't have an MDP to begin with anymore. So this was NOT PLANNING.

What Just Happened?

We interacted with the real world. Then we took those observations and used them to figure out a little bit more information about the MDP. And as we gathered more information about the MDP we started to have different opinions as to what we should. Red no longer seemed like the good policy.

- 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

We needed to act to approximate the parameters of the system! As you act more, the better that approximation becomes

sometimes you have to explore for the experience/ information, not the yield. That experience helps you make better decisions in the future.

- 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

regret is the difference between the utility of *your* optimal policy, and the utility of the *actual* optimal policy. So there's zero regret if we know the MDP parameters

you can't just pull the red lever once and determine that you'll always get that reward. You have to sample a lot to get a better estimate.

it's much harder to learn a MDP than solve a known MDP

Next Time: Reinforcement Learning!

RL. In the next lecture we'll think about how we should act when there are MDPs but we don't know any of the parameters and the only way to know what's going on is to interact with the world