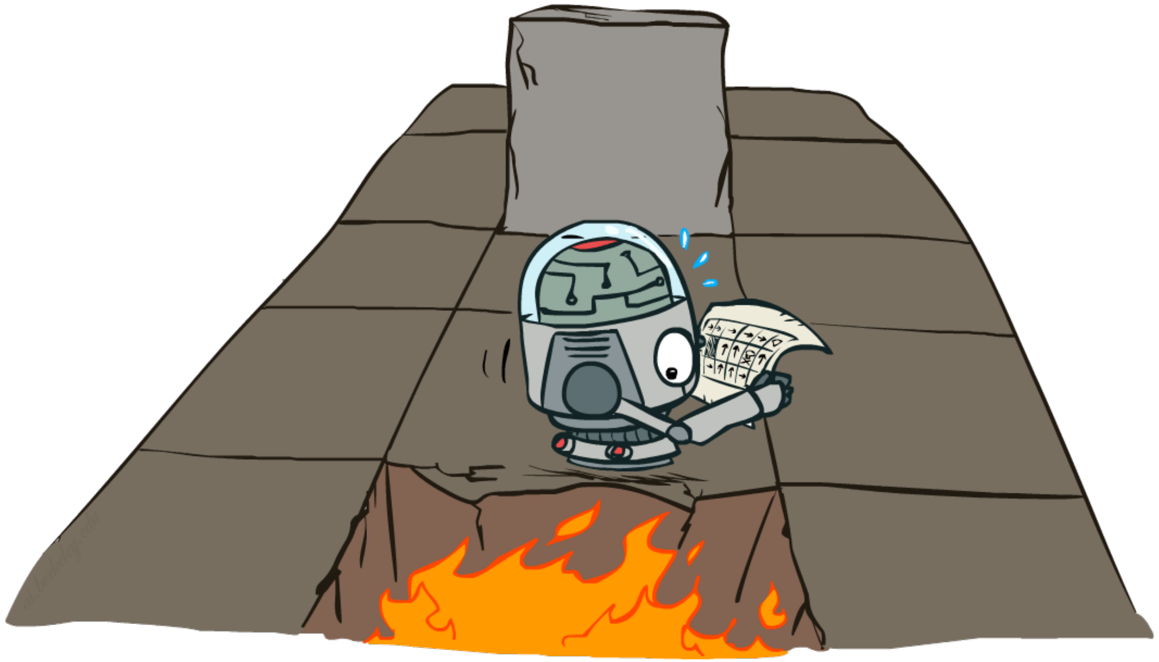
CS 4100 // artificial intelligence

instructor: byron wallace



Markov Decision Processes (MDPs) II

Attribution: many of these slides are modified versions of those distributed with the *UC Berkeley CS188* materials
Thanks to John DeNero and Dan Klein

Last time: grid world

A maze-like problem

- The agent lives in a grid
- Walls block the agent's path

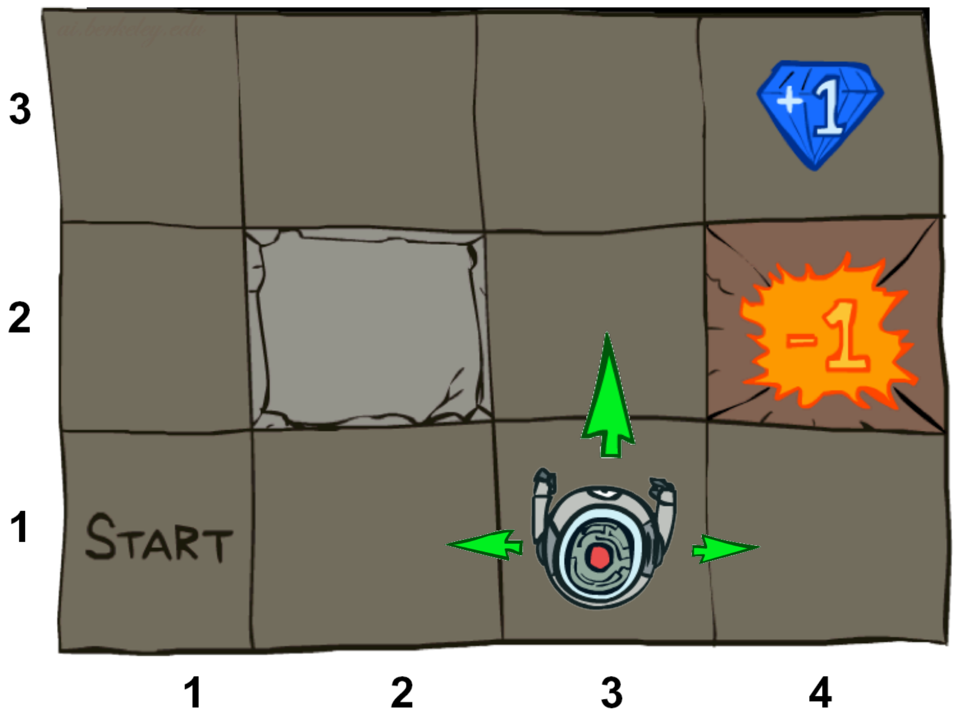
Noisy movement: actions do not always go as planned

- 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

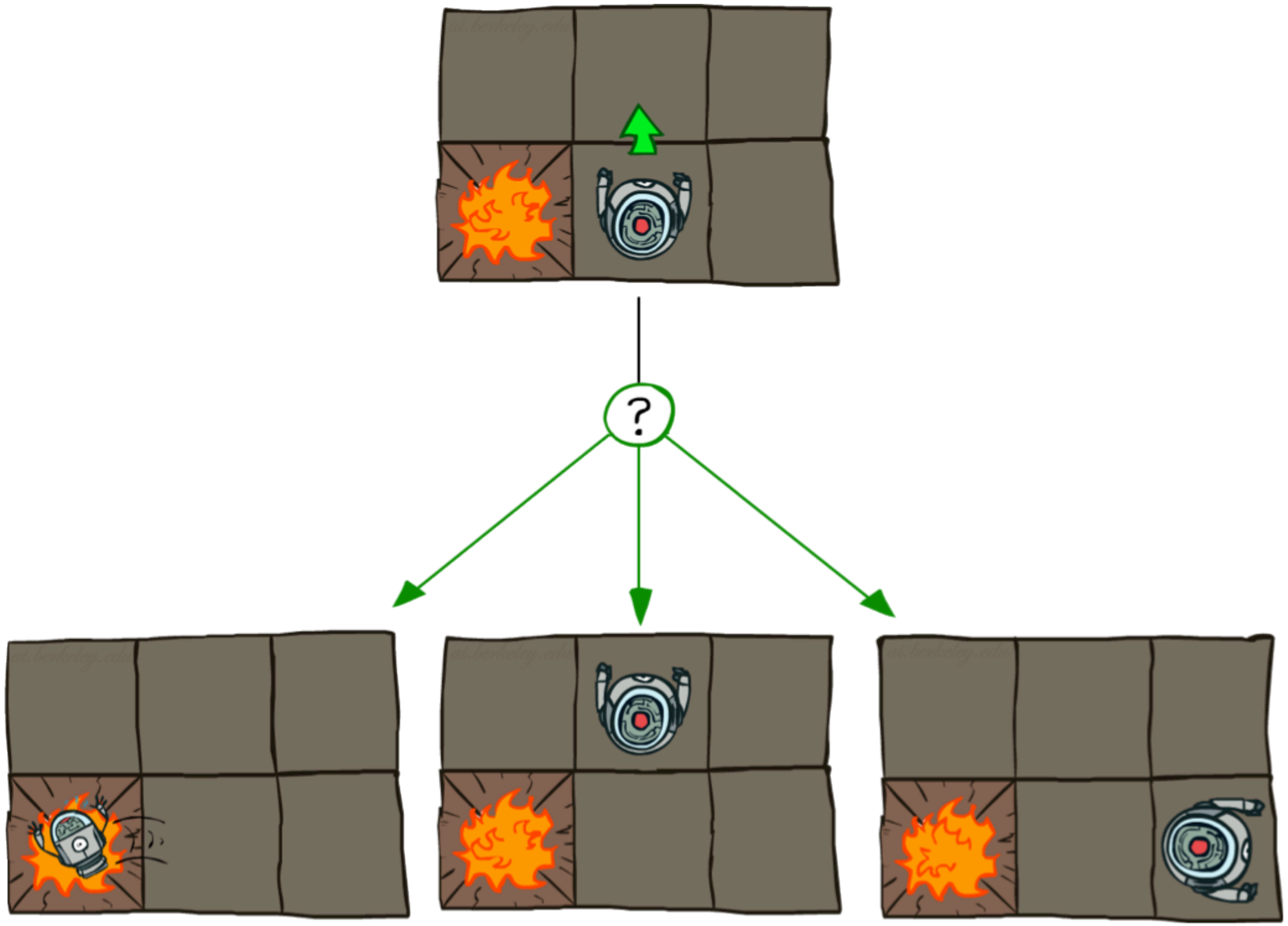
The agent receives rewards each time step

- Small "living" reward each step (can be negative)
- Big rewards come at the end (good or bad)

Goal: *maximize sum of rewards*



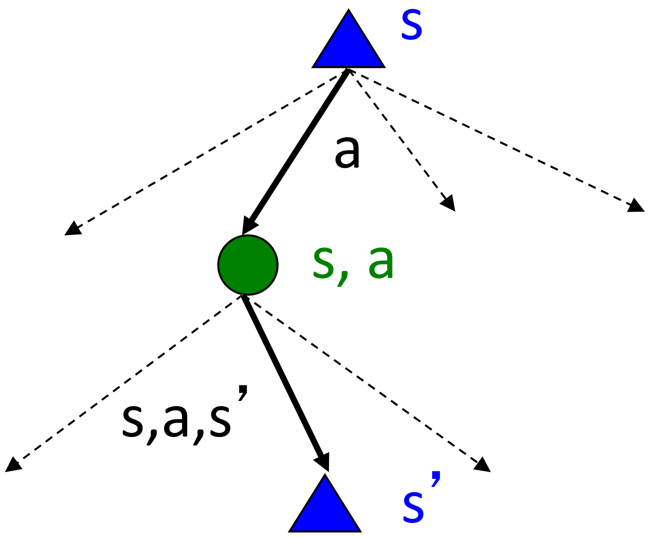
Grid world is *stochastic*



Review: Markov Decision Processes (MDPs)

An MDP is defined by

- States $\in$ S
- Actions a $\in$ 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
- Reward function R(s, a, s') and discount γ
  Sometimes just R(s) or R(s')
- Start state
- Maybe a terminal state



Quantities

- Policy = map of states to actions
- Utility = sum of discounted rewards
- Values = *expected* future utility from a state, under optimal action
- Q-Values = expected future utility from a q-state (chance node)

Optimal quantities

The value (utility) of a state s

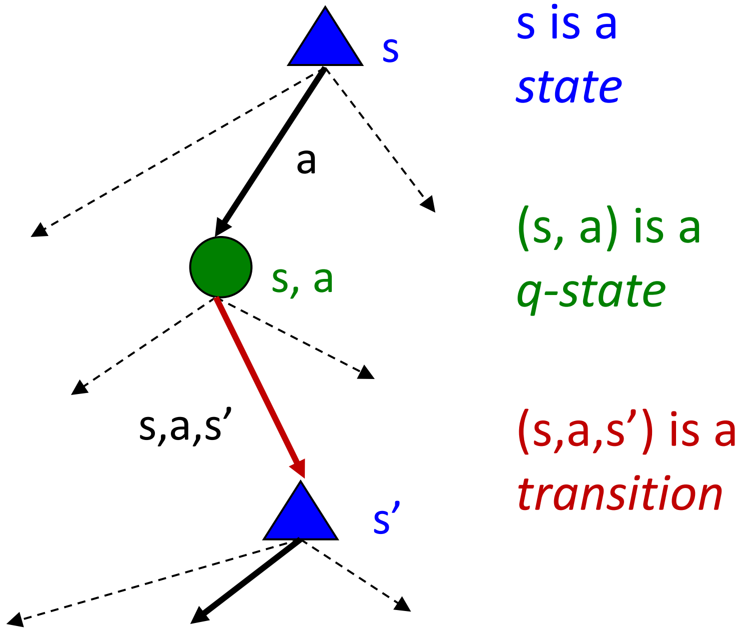
$V^*(s)$ = *expected utility* starting in s and acting optimally. Note: sometimes written as U(s)

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



Policies

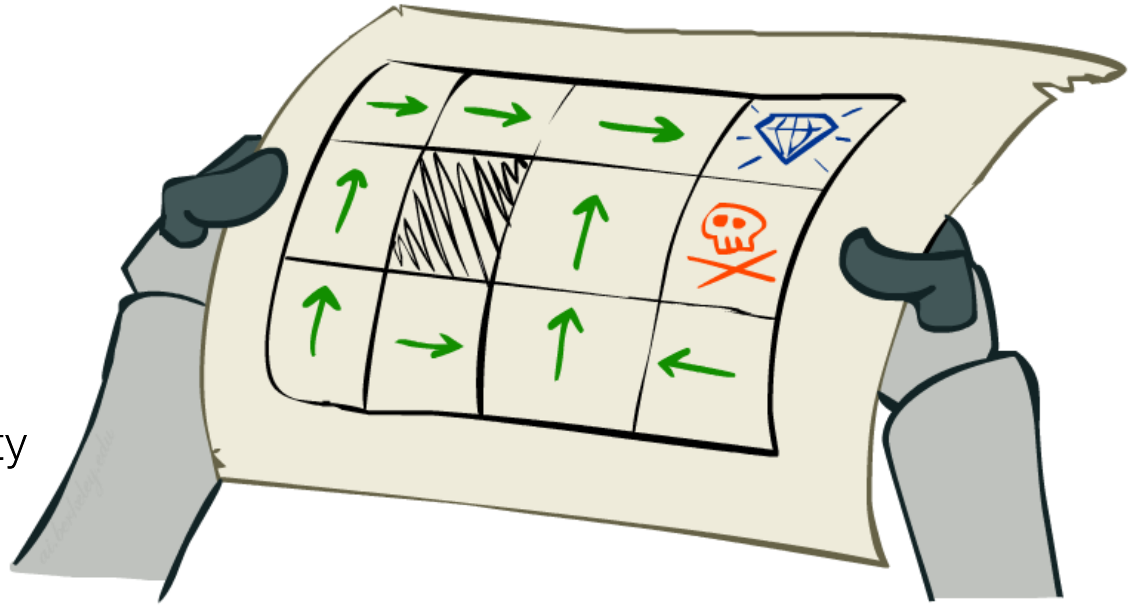
In deterministic single-agent search problems, we wanted an optimal **plan**, or sequence of actions, from start to a goal

For MDPs, we want an optimal policy π^*: S → 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

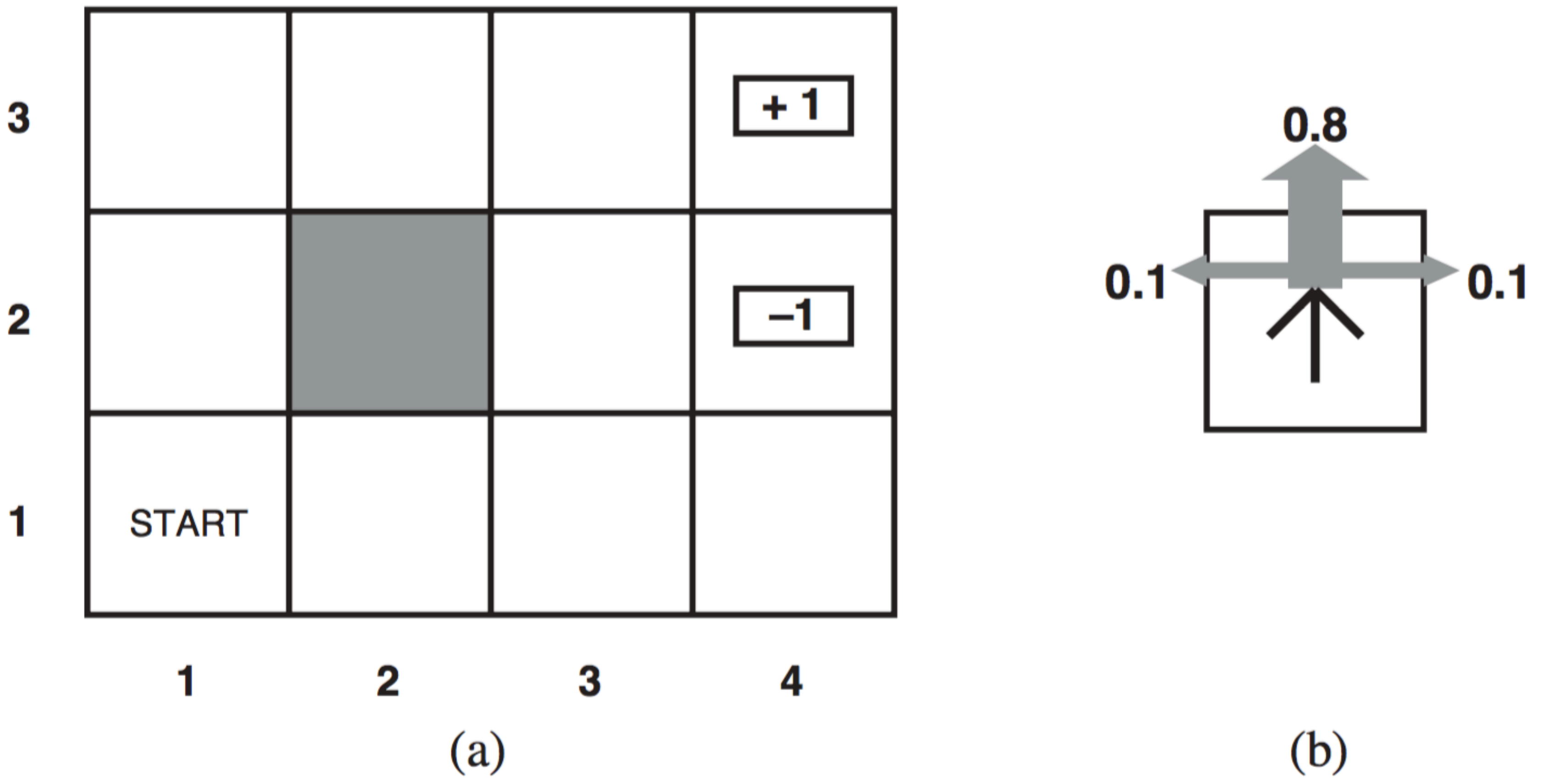
Expectimax didn't compute entire policies

- It computed the action for a single state only



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

Gridworld



Gridworld

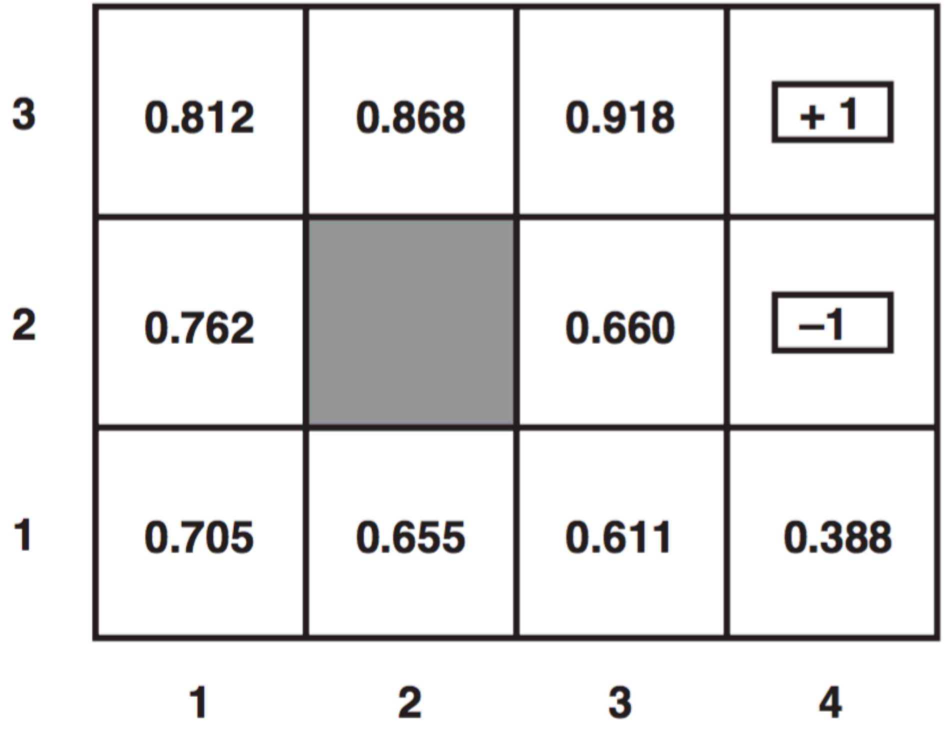


Figure 17.3 The utilities of the states in the 4×3 world, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

Gridworld

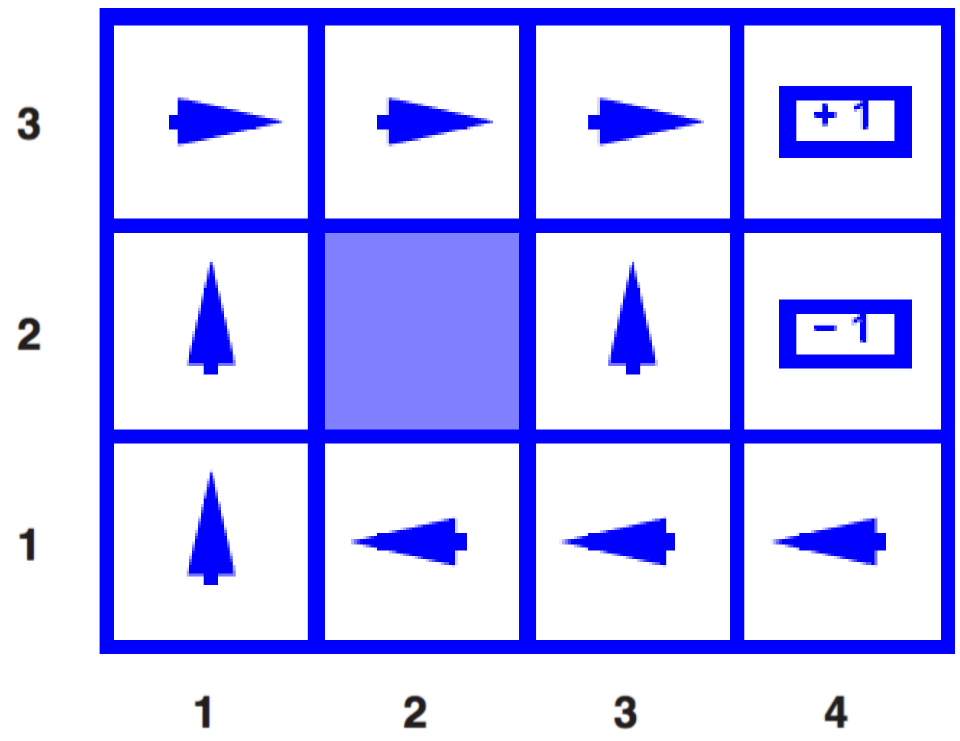
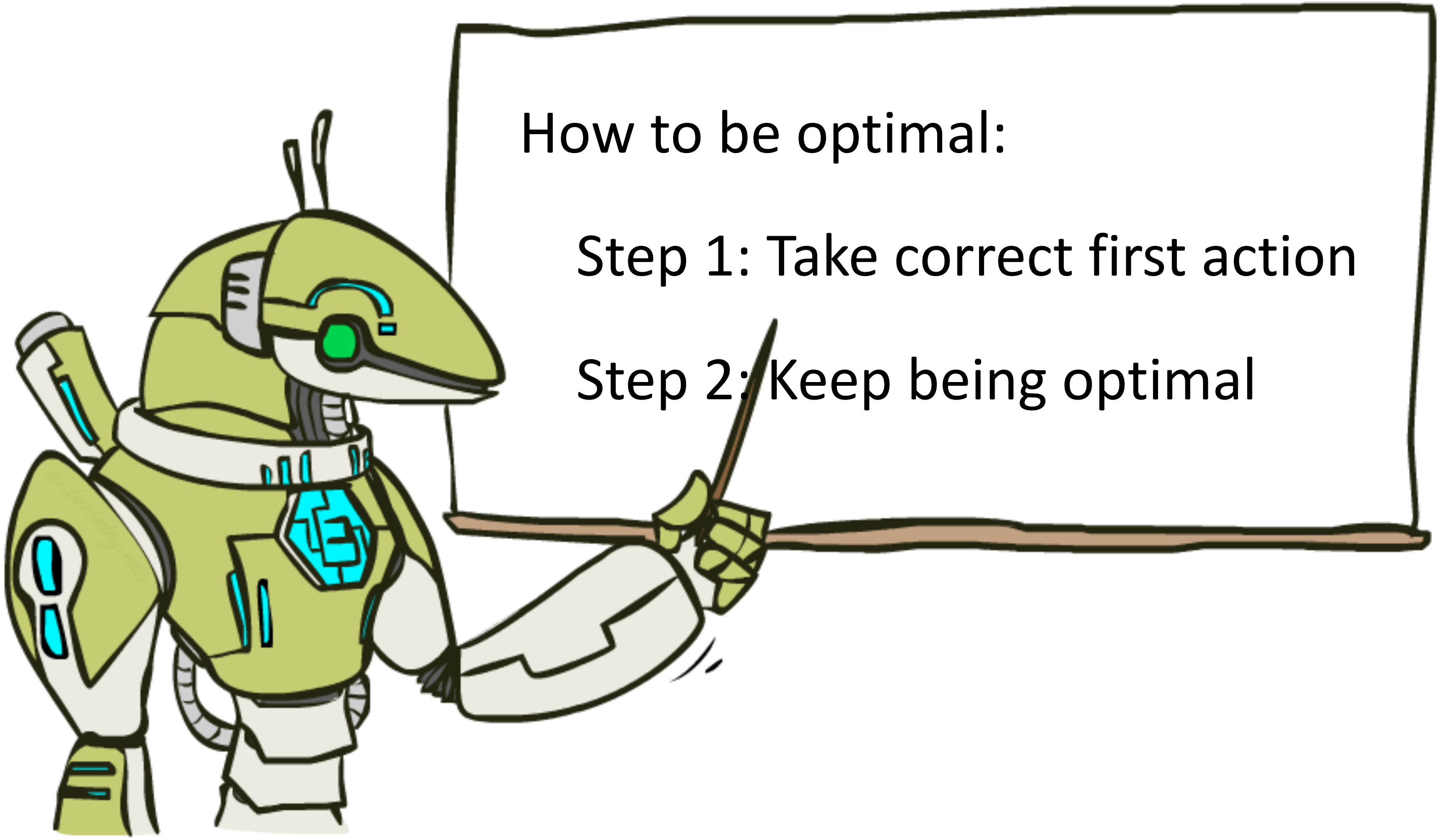


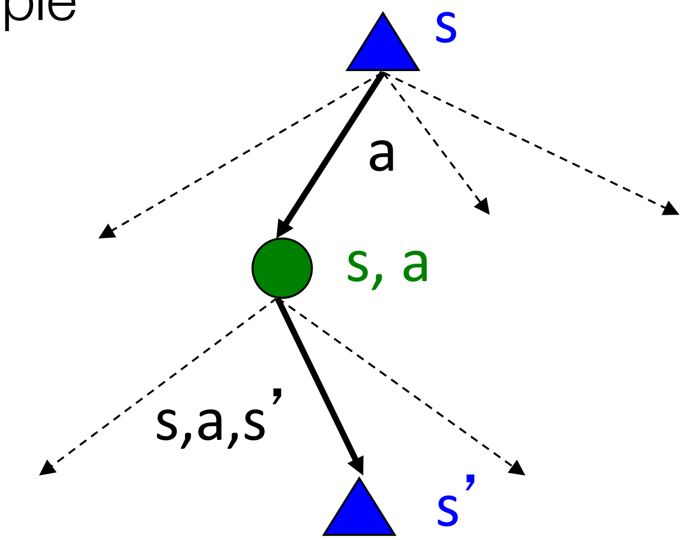
Figure 17.3 The utilities of the states in the 4×3 world, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

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

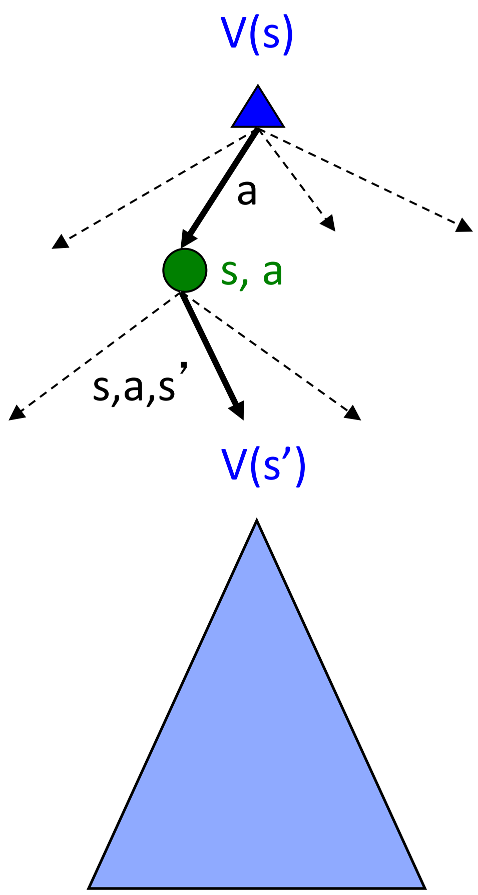
Bellman equations characterize the optimal values:

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

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 an *iterative solution method*



Exercise from last time

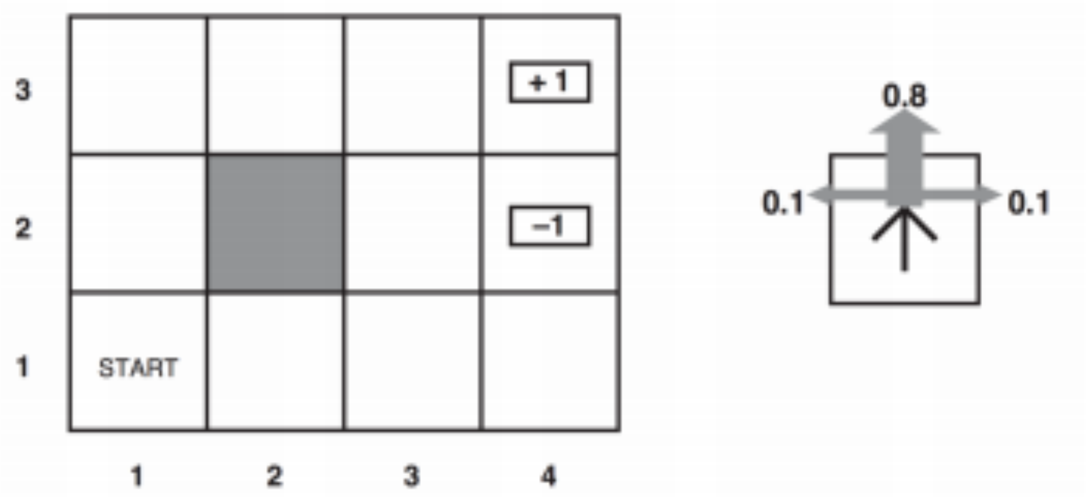
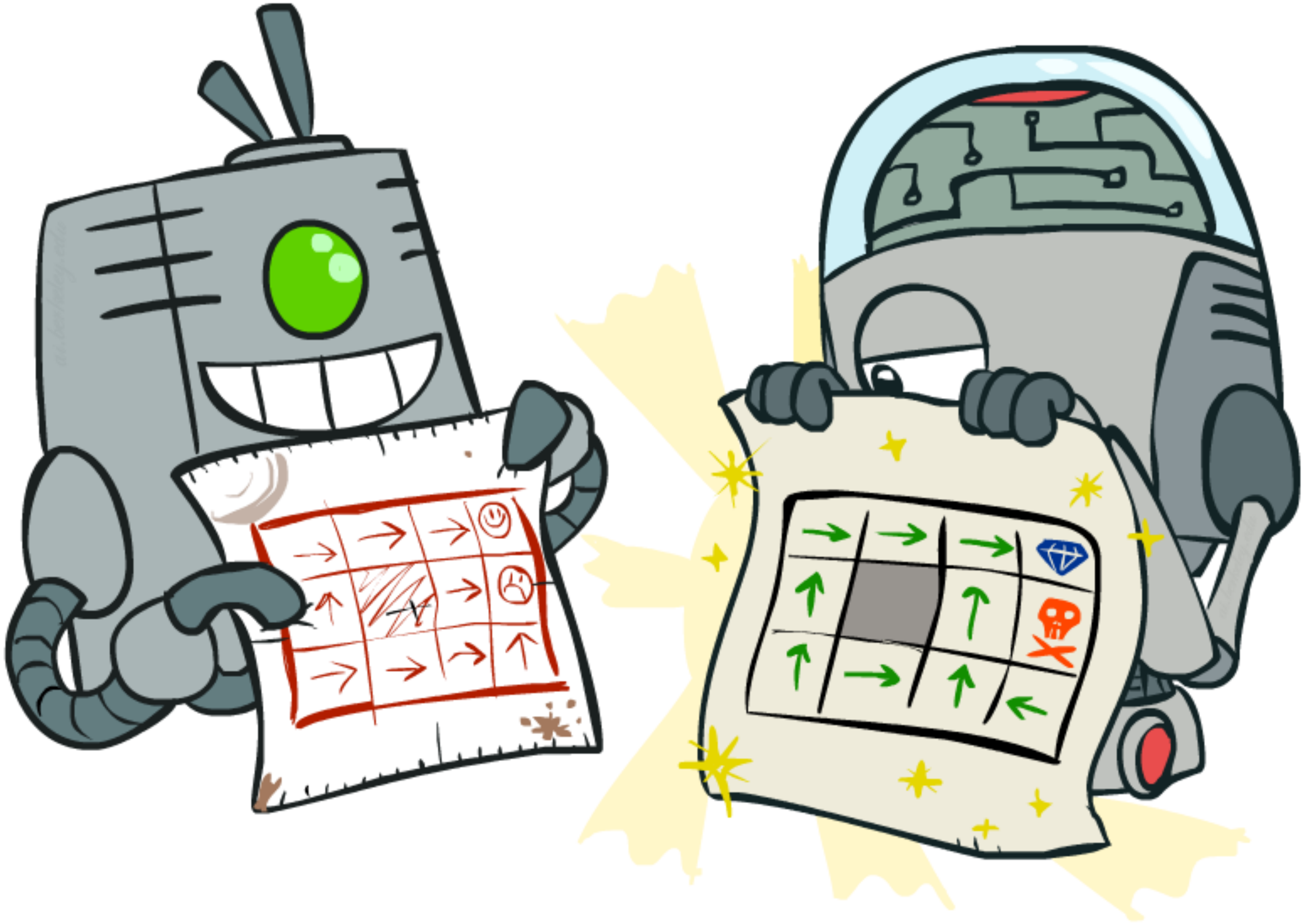


Figure 1: Exciting gridworld from the text (Figure 17.1). Assume $R = -0.3$ (i.e., the 'living penalty' is -0.3).

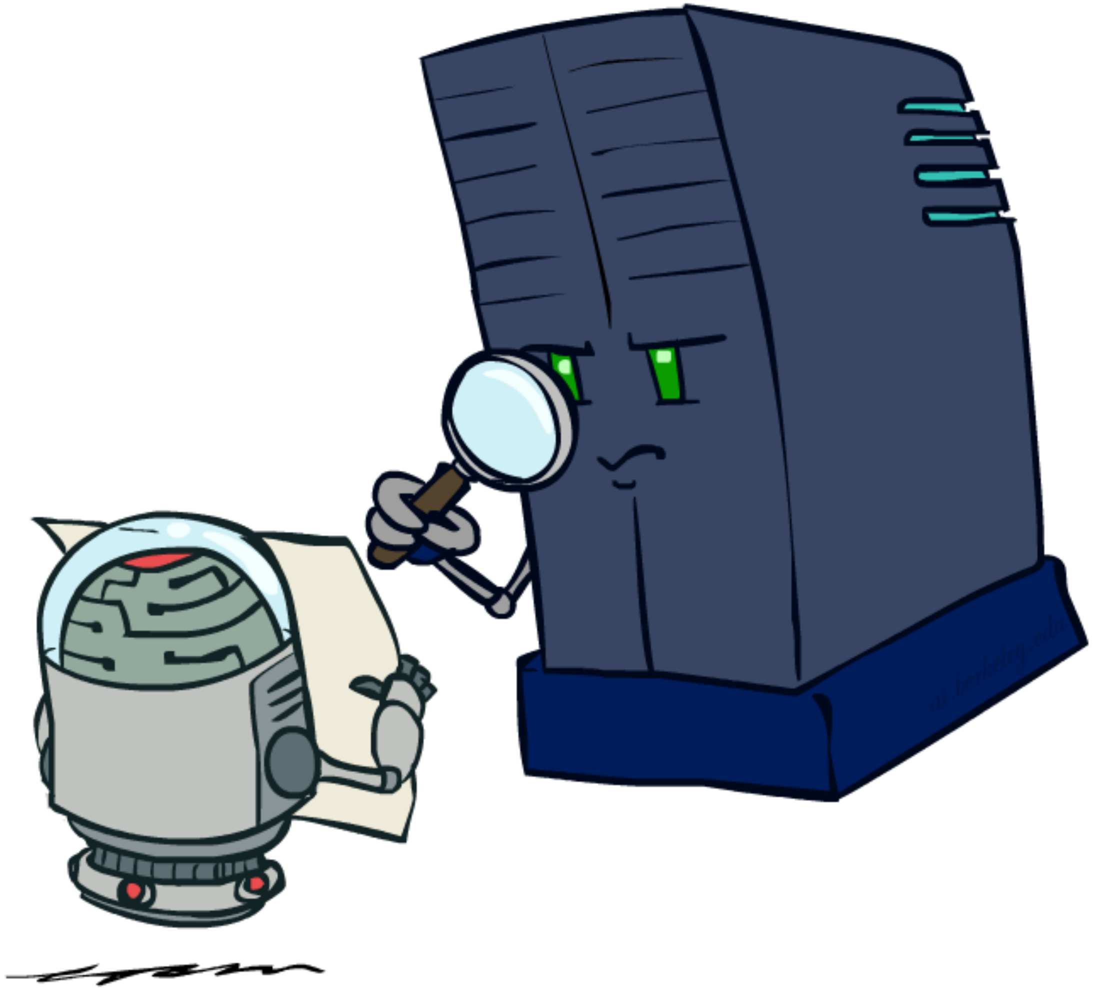
Remember: calculate by adding the *instantaneous reward* at a state to the expected utility that will be achieved by the best possible following sequence of actions.

Policies



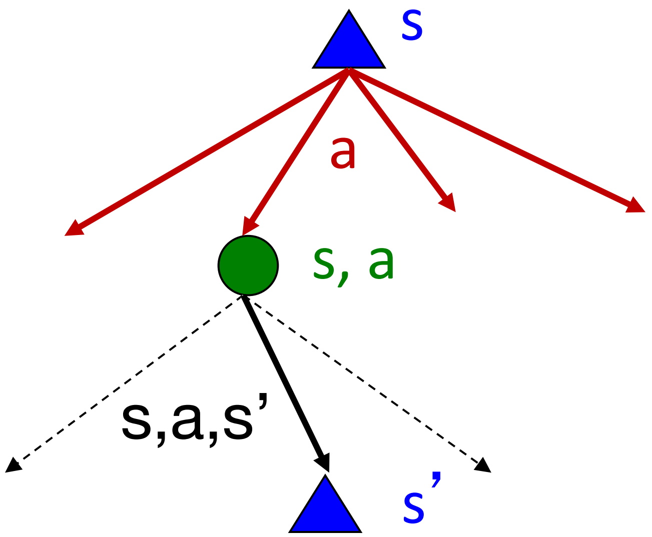
Policy evaluation

Need a means of evaluating a given policy

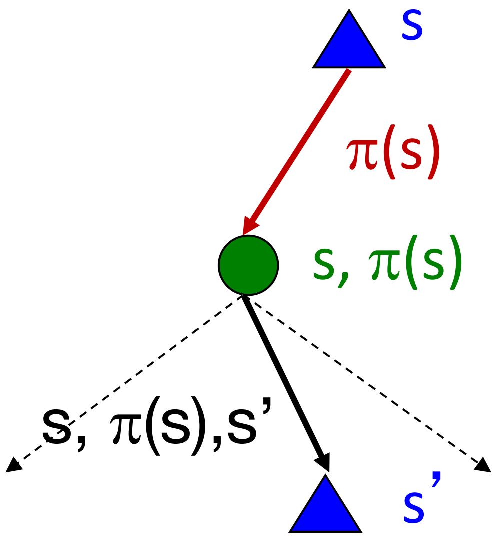


Fixed policies

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

Utilities for a fixed policy

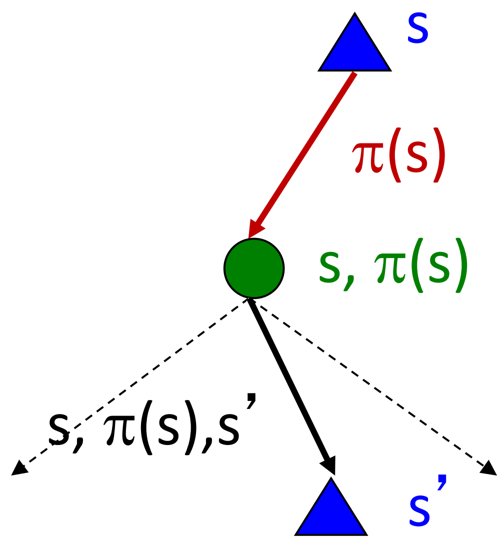
- Basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- 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

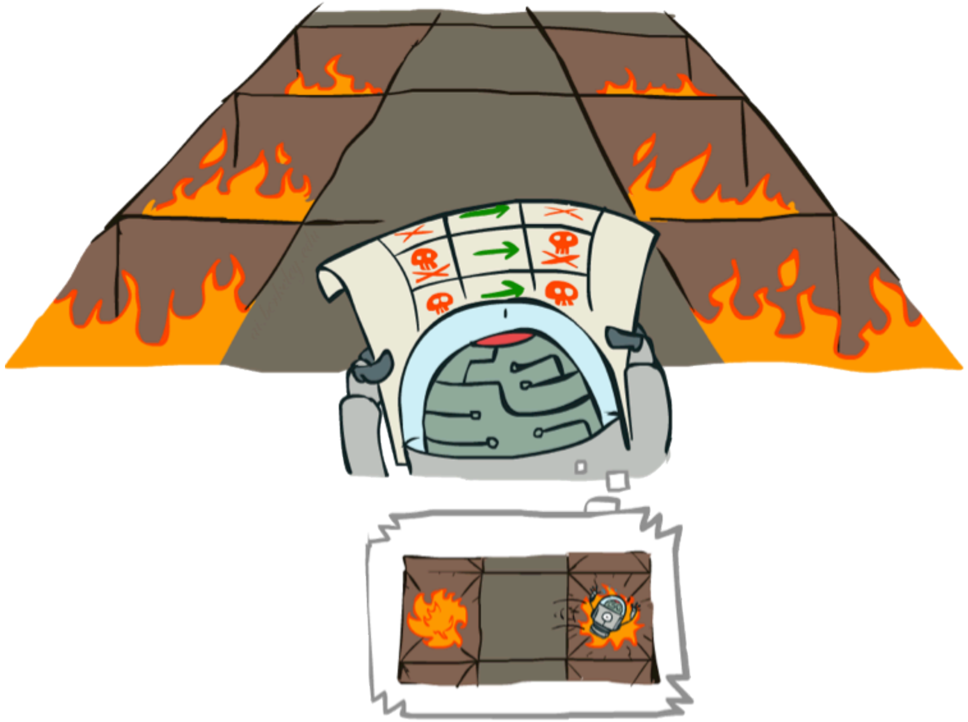
- Recursive relation (one-step look-ahead / Bellman equation):

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

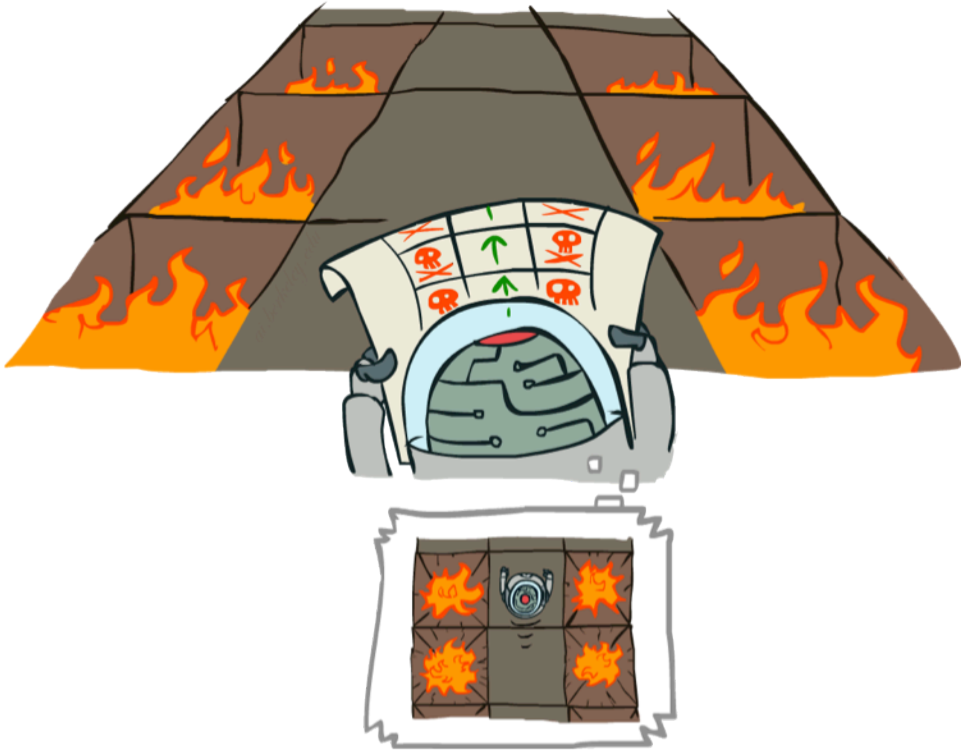


Example: policy evaluation

Always Go Right



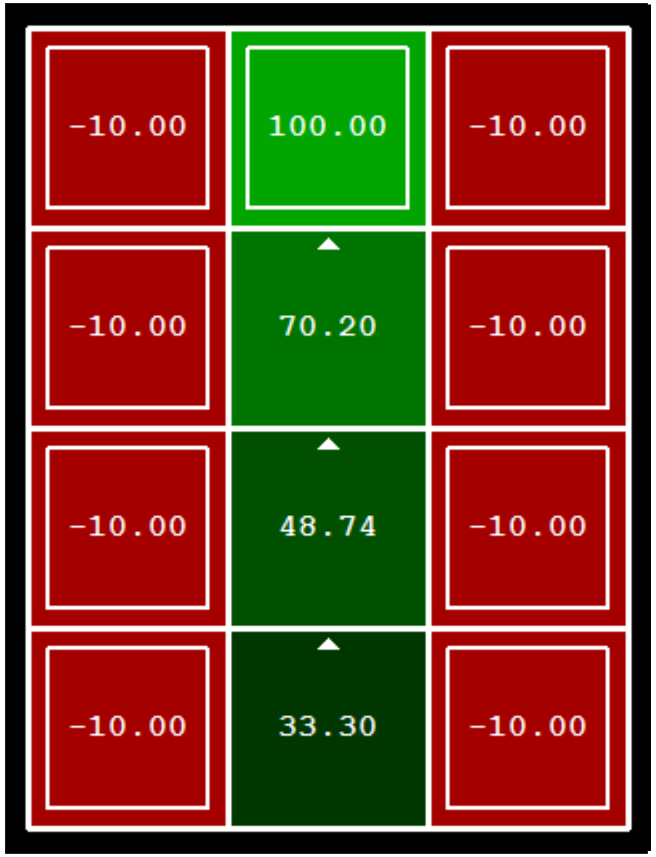
Always Go Forward



Example: policy evaluation

Always Go Right

Always Go Forward



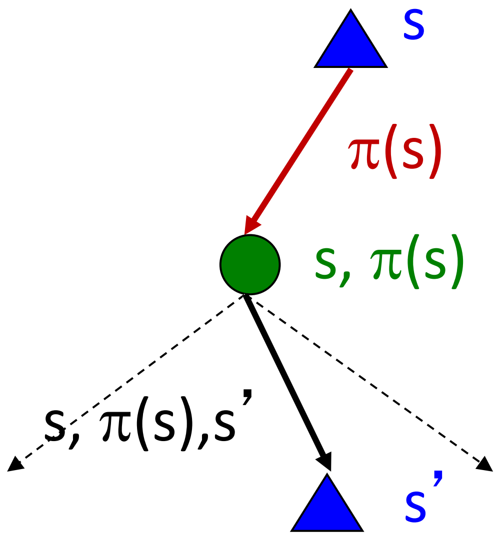
Policy evaluation

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

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

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

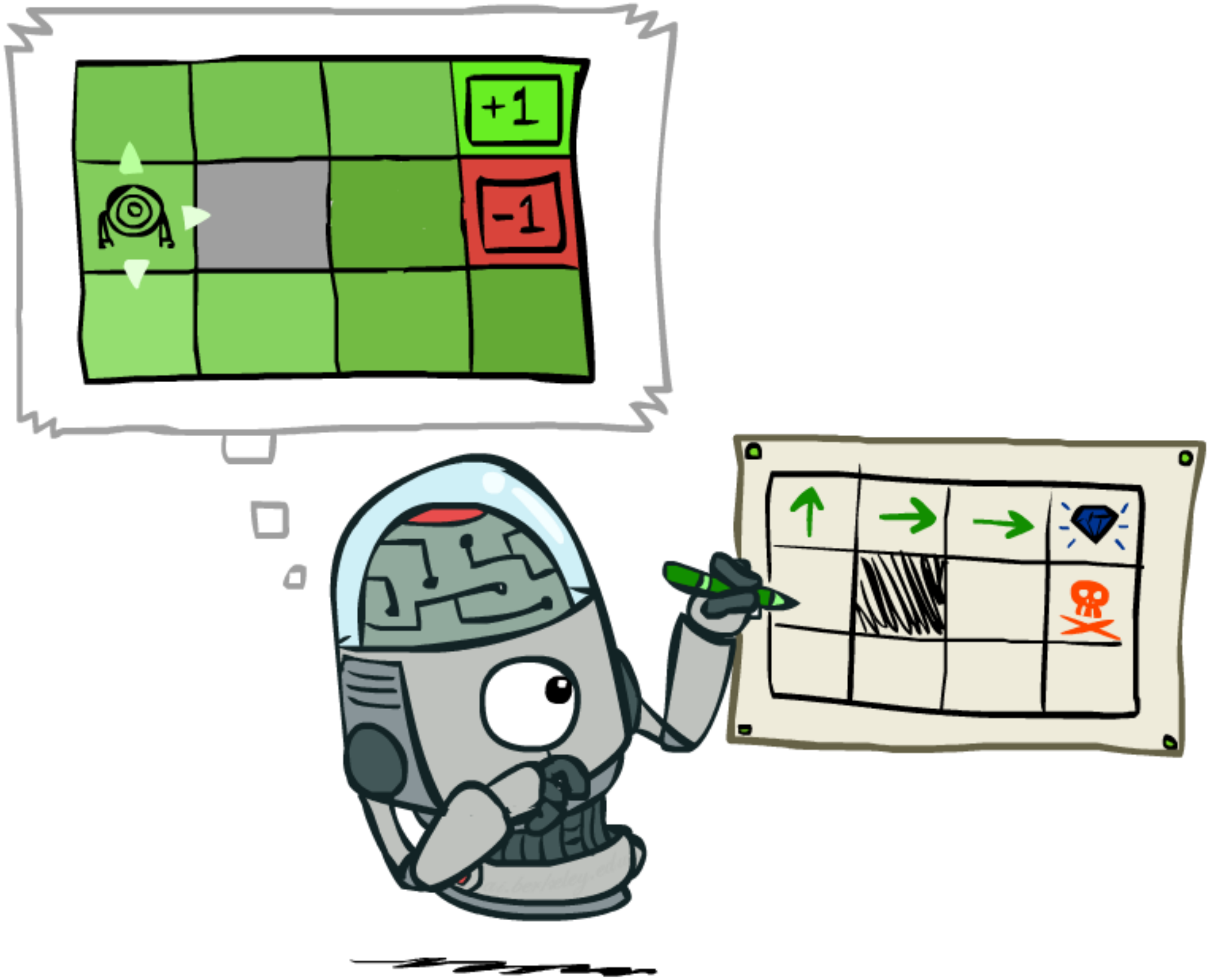
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



- Efficiency: $O(S^2)$ per iteration (we get to drop the a)

- Note that the maxes are gone, so the Bellman equations are just a linear system
  Could solve with Matlab (or your favorite linear system solver)

Policy extraction



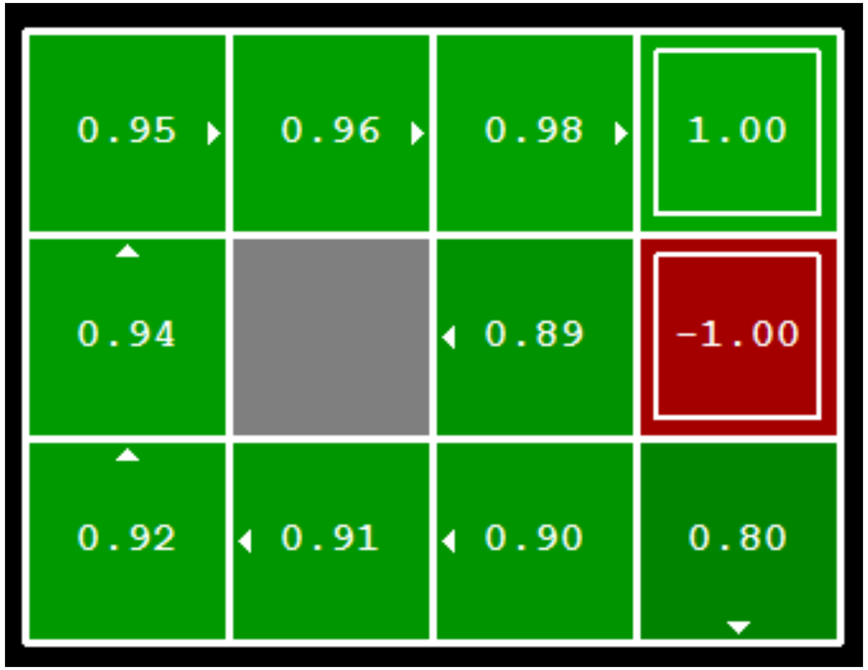
Computing actions from values

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)

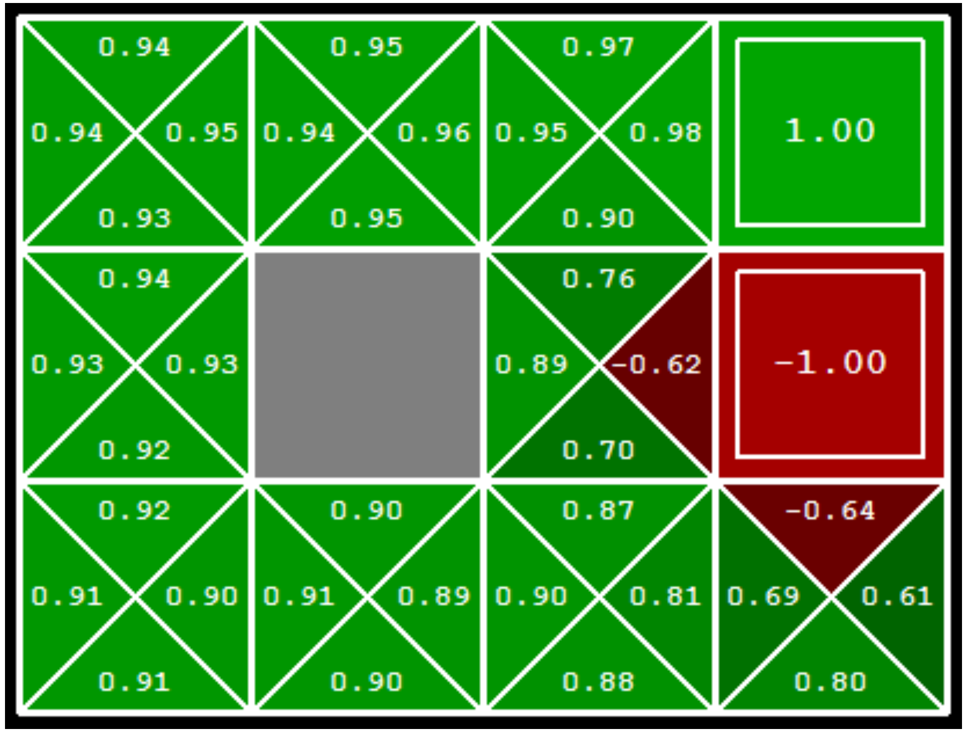


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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:



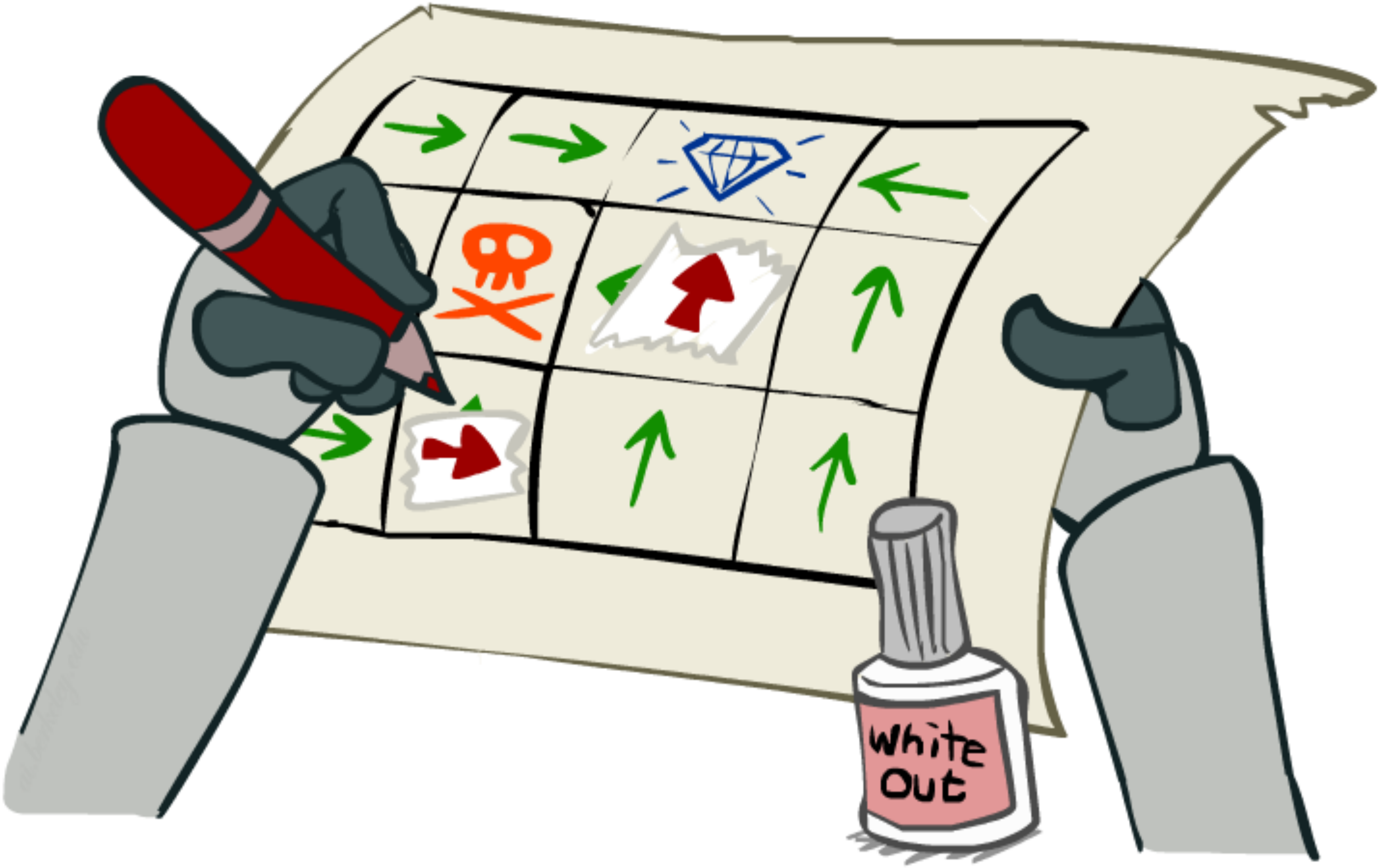
How should we act?

- Completely trivial to decide!

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

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

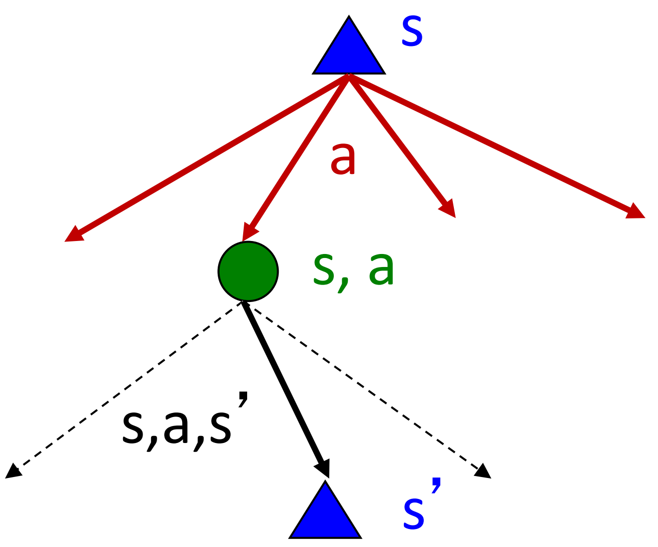
Policy iteration



Problems with value Iteration

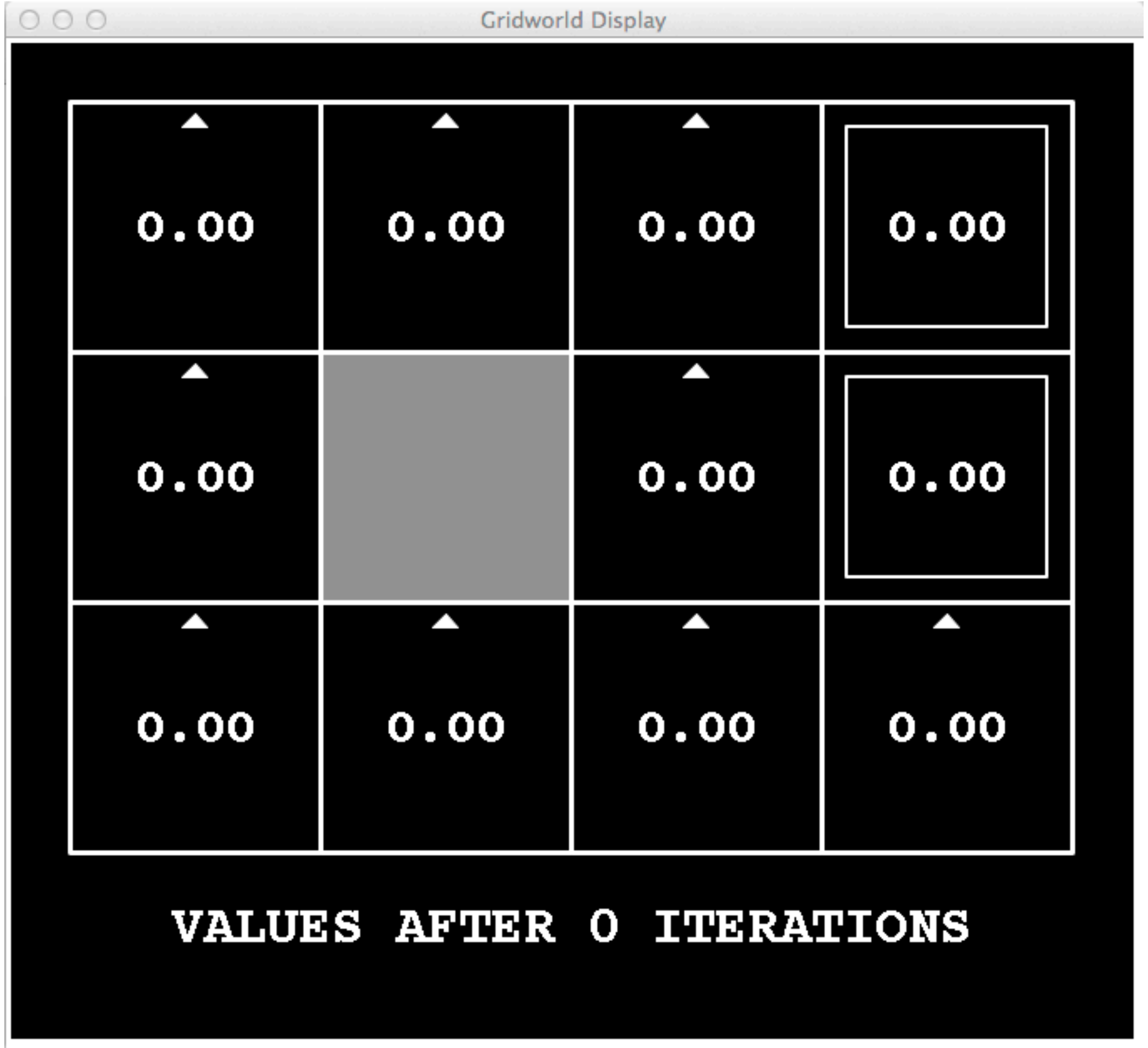
Value iteration repeats the Bellman updates:

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



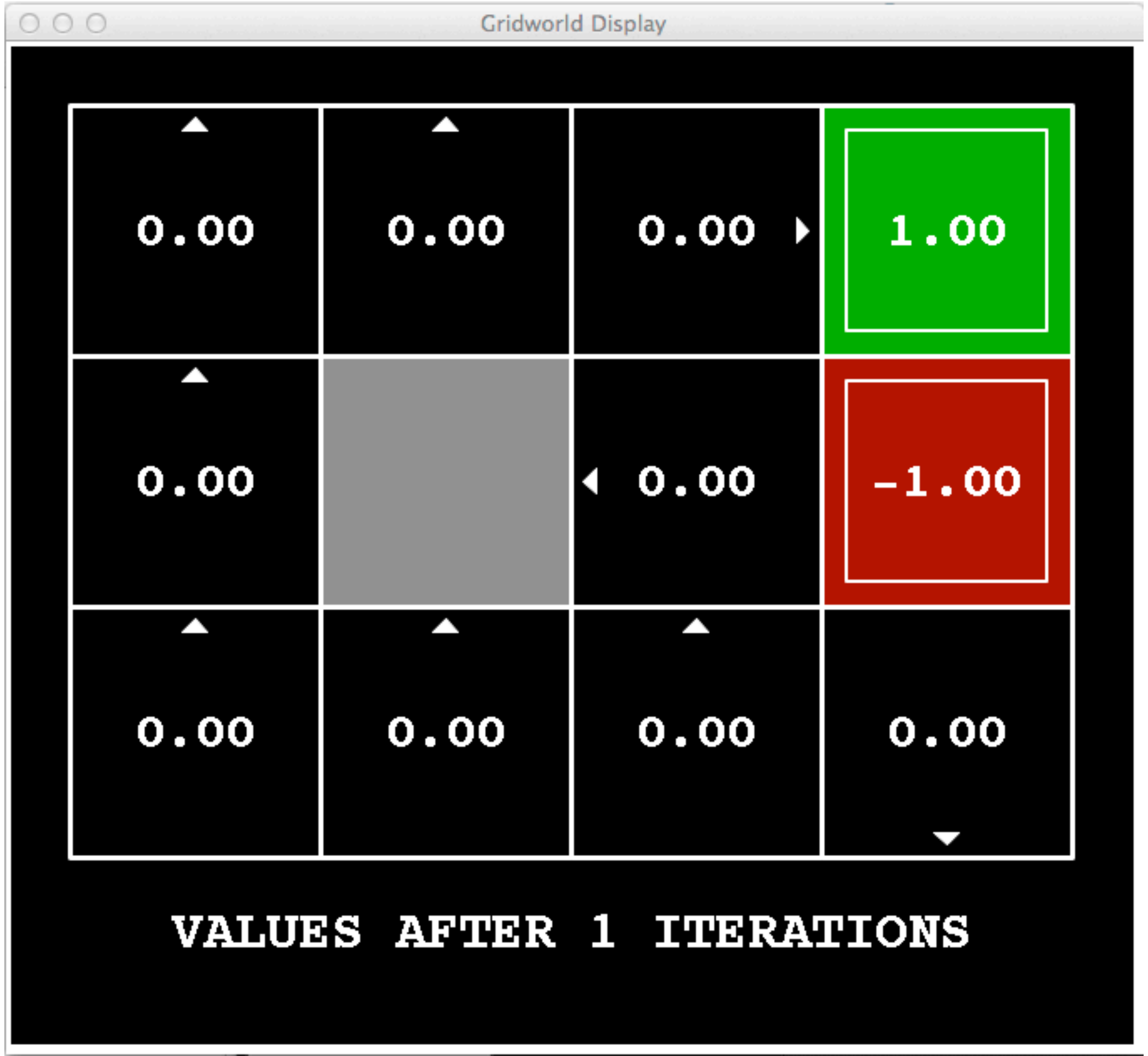
- Problem 1: It’s slow – $O(S^2A)$ per iteration
- Problem 2: The “max” at each state rarely changes
- Problem 3: The policy often converges long before the values

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

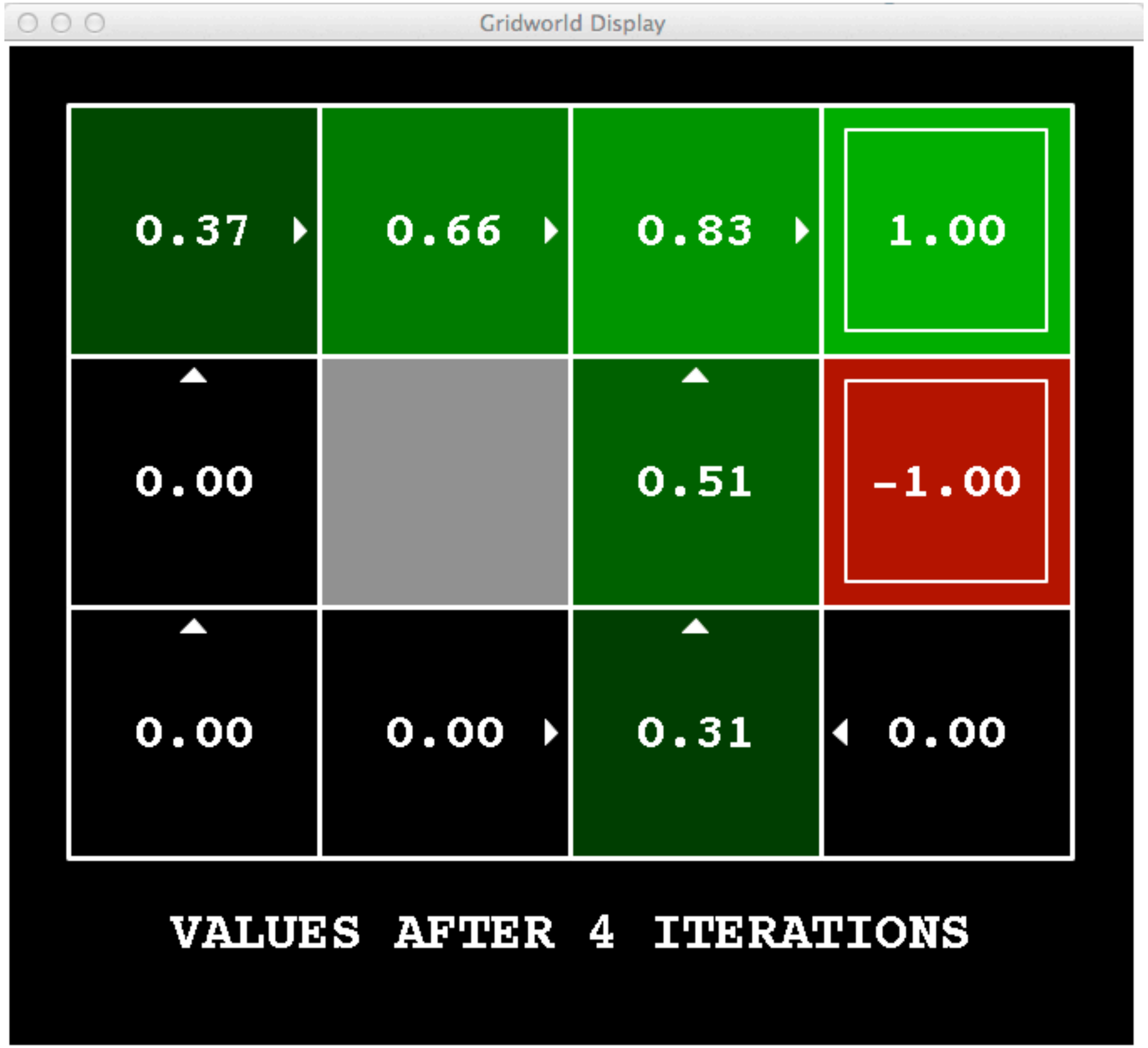
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

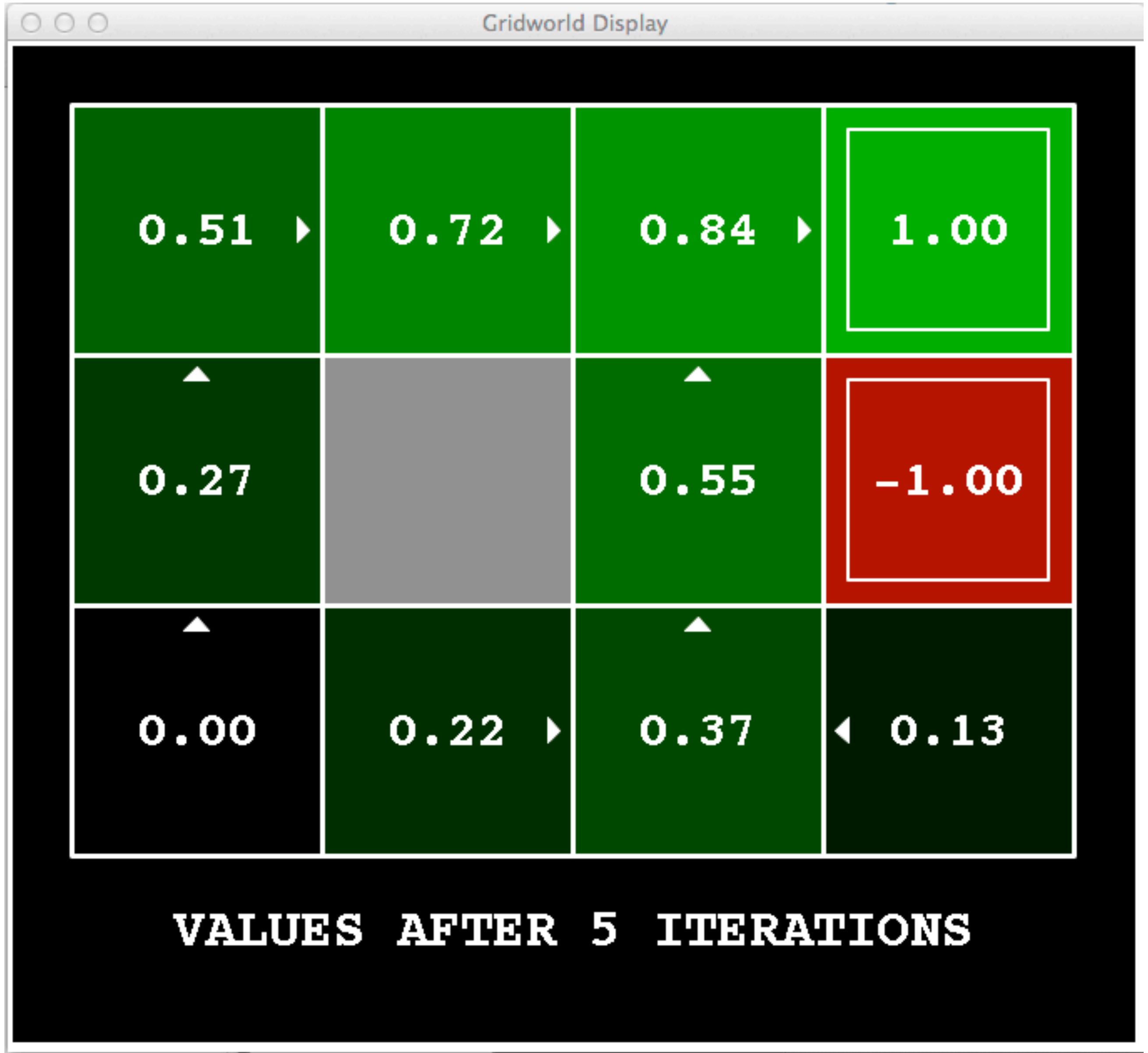
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



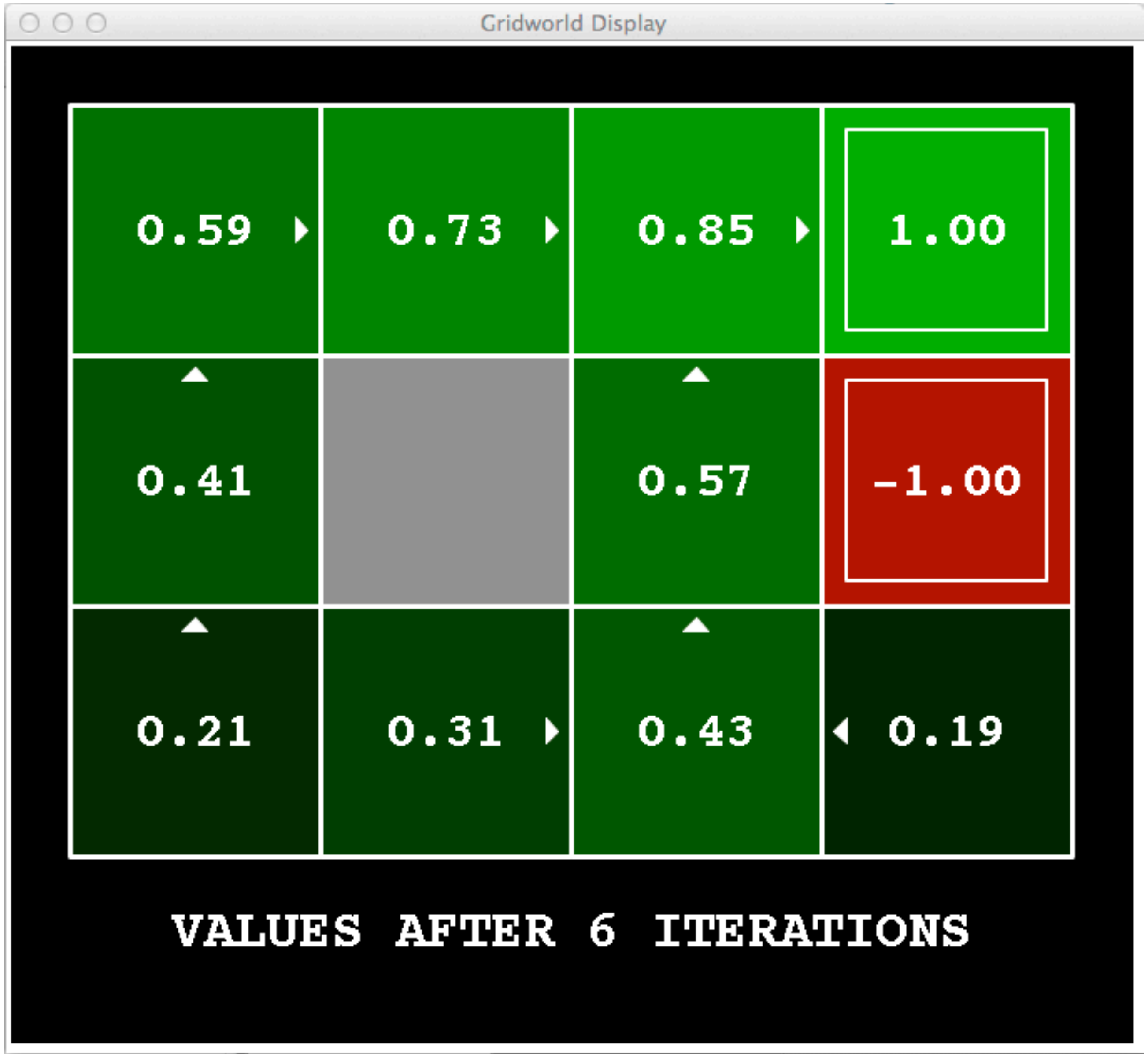
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



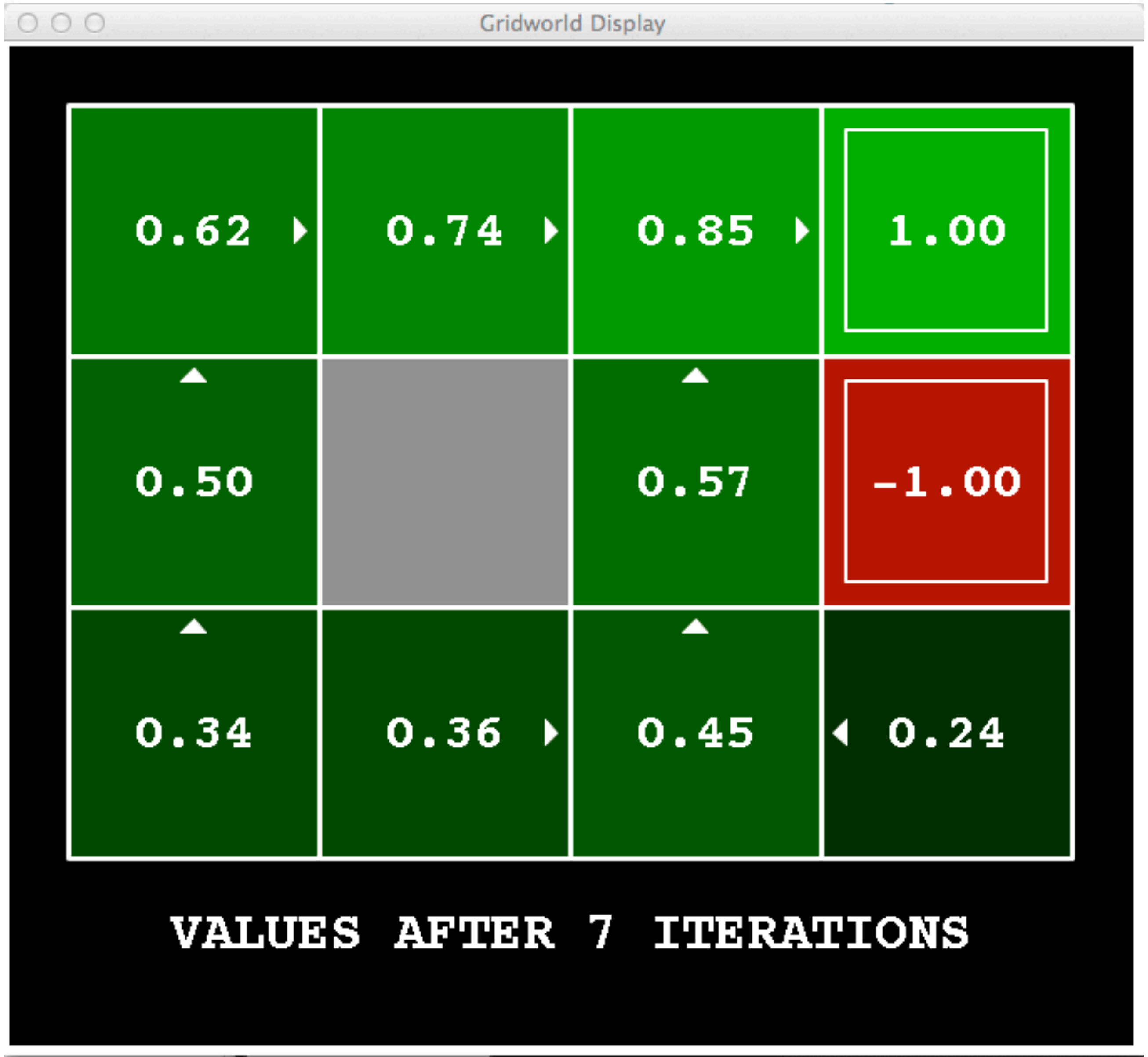
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



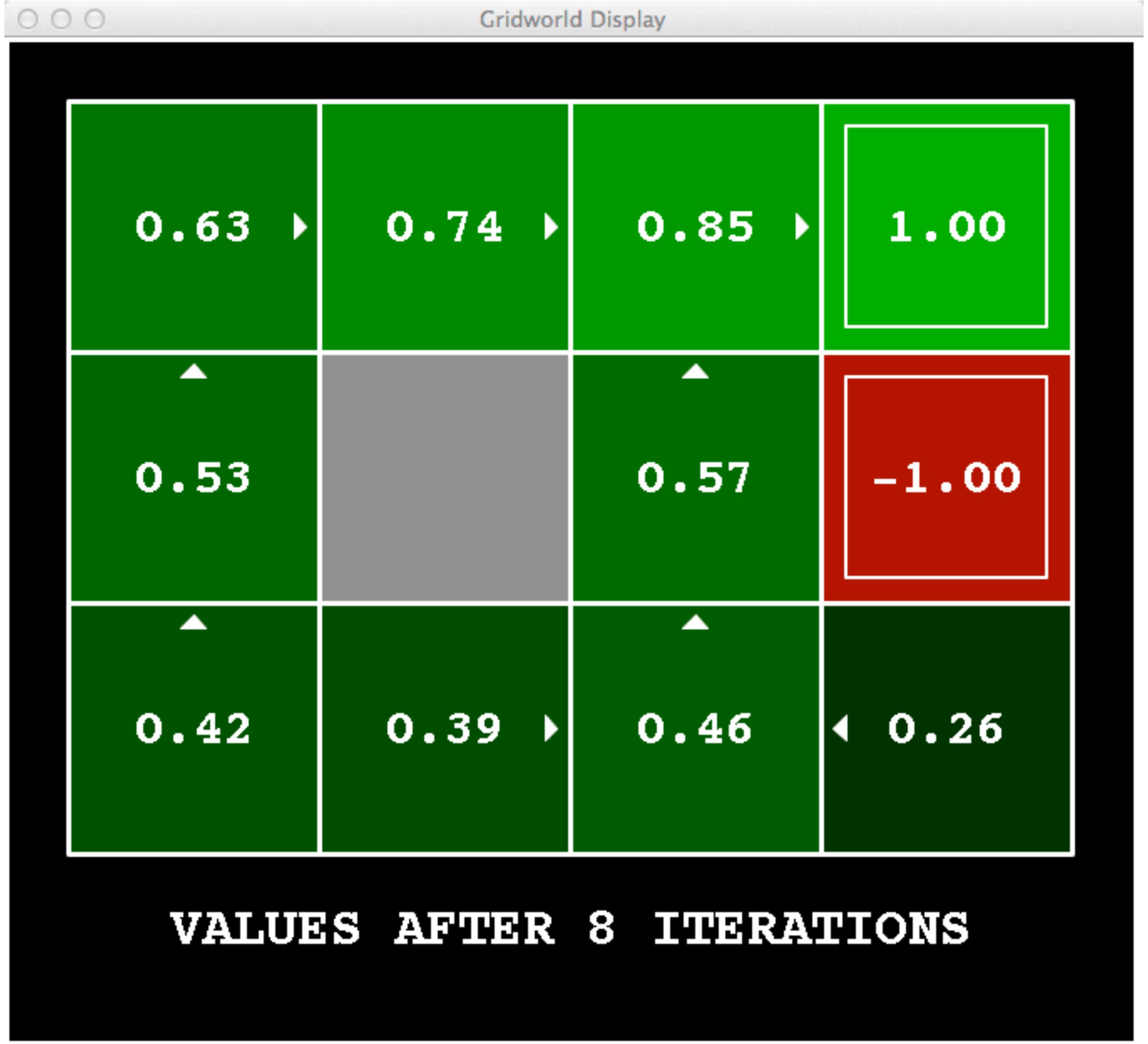
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



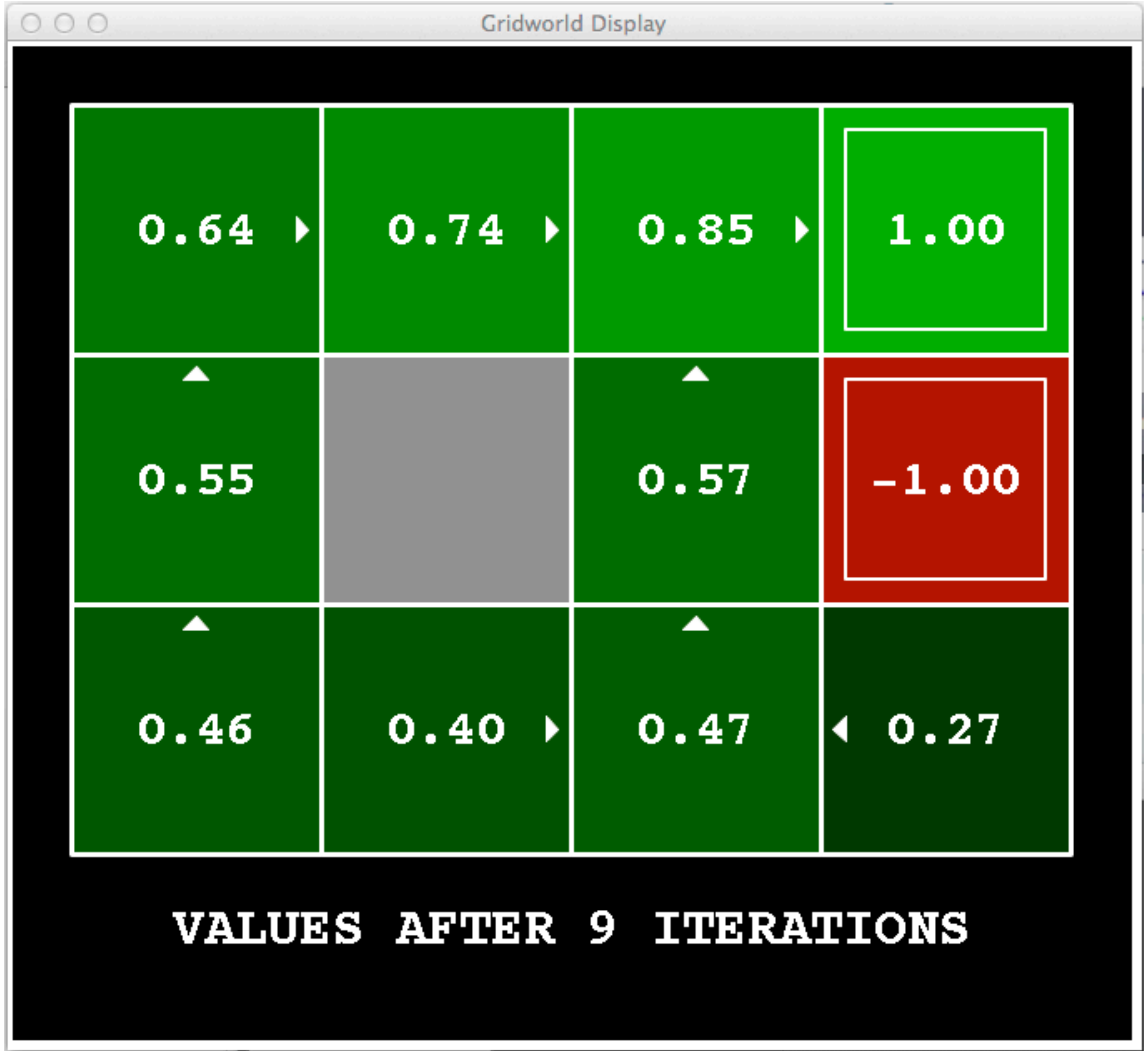
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



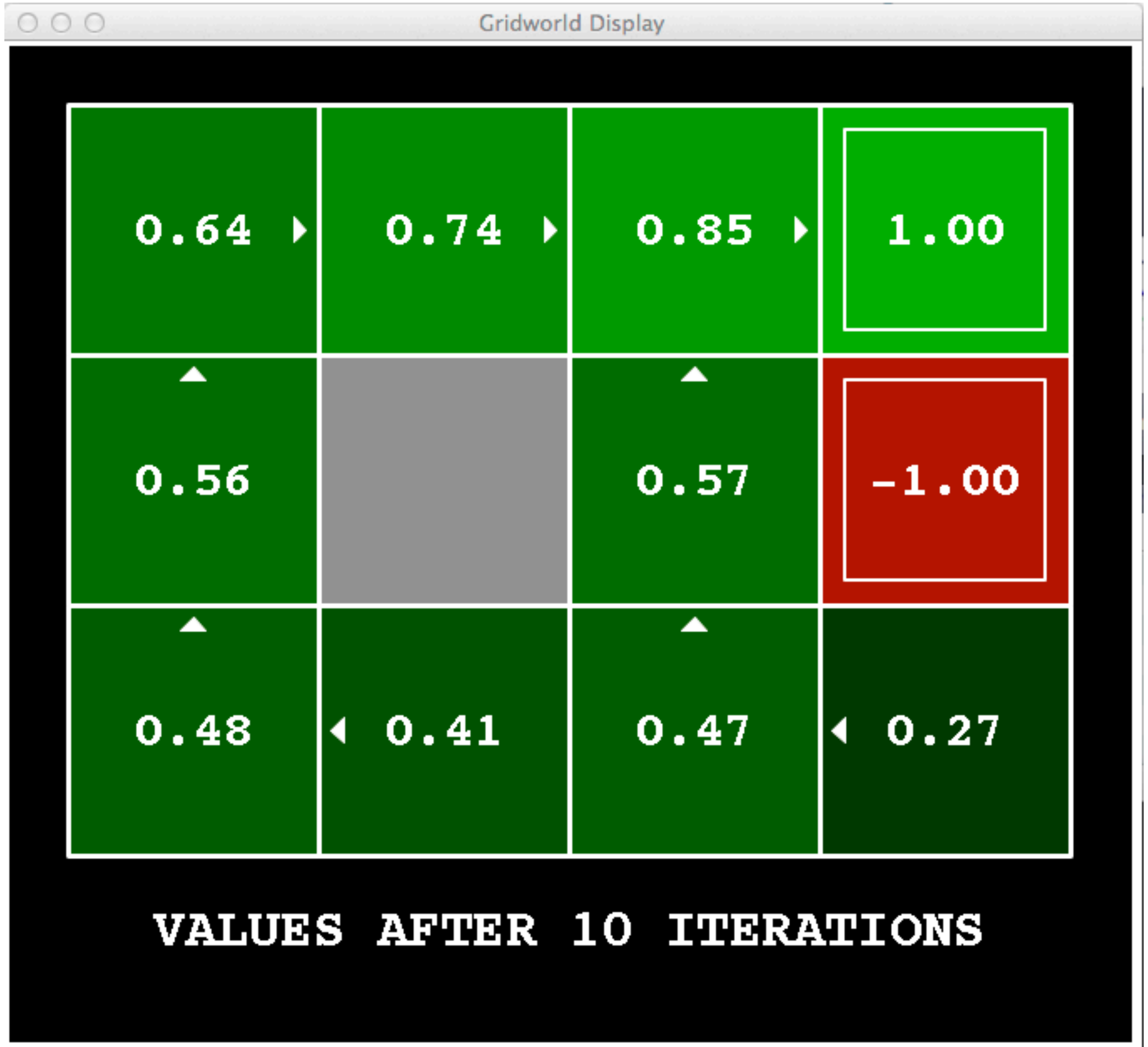
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



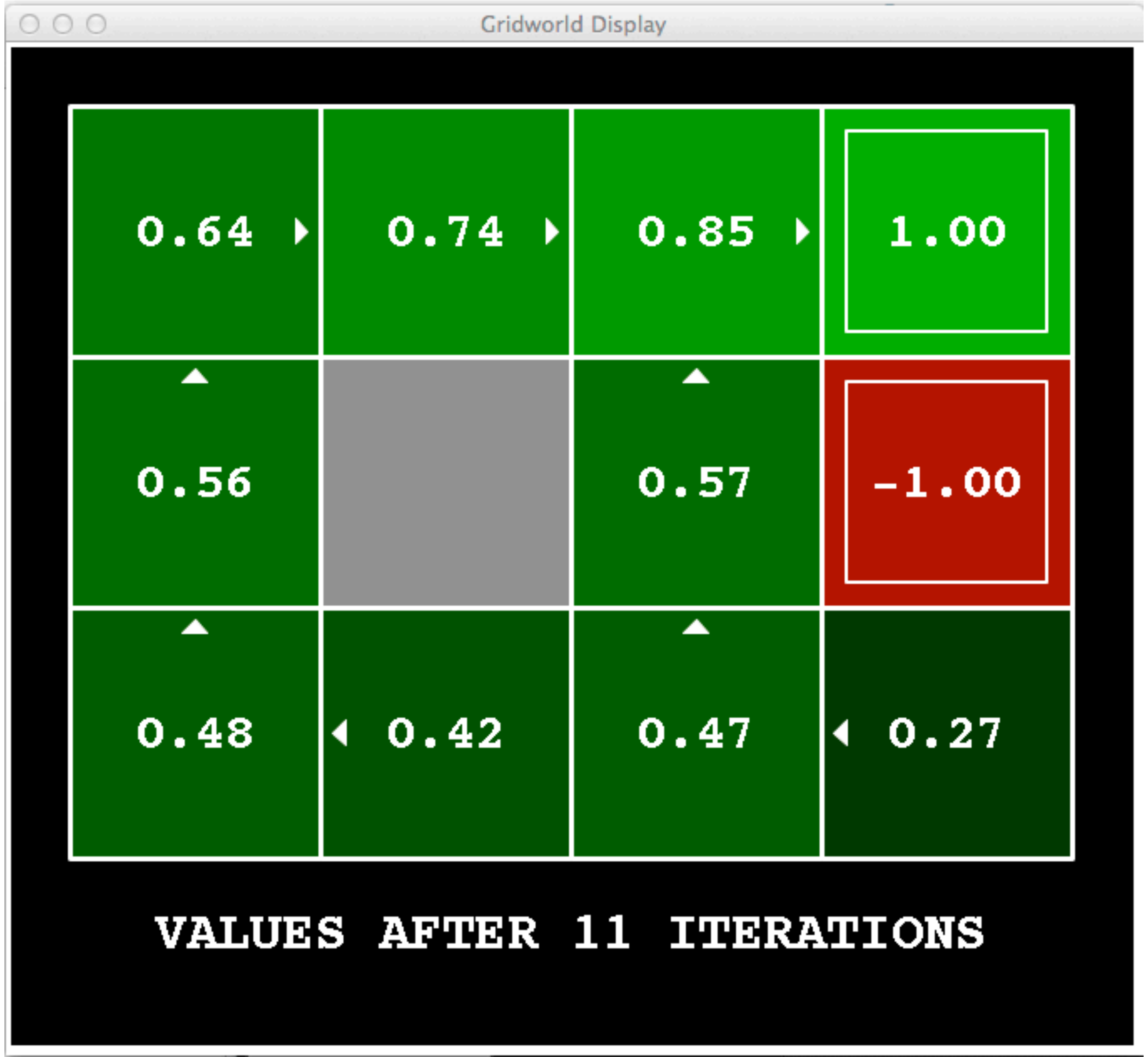
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



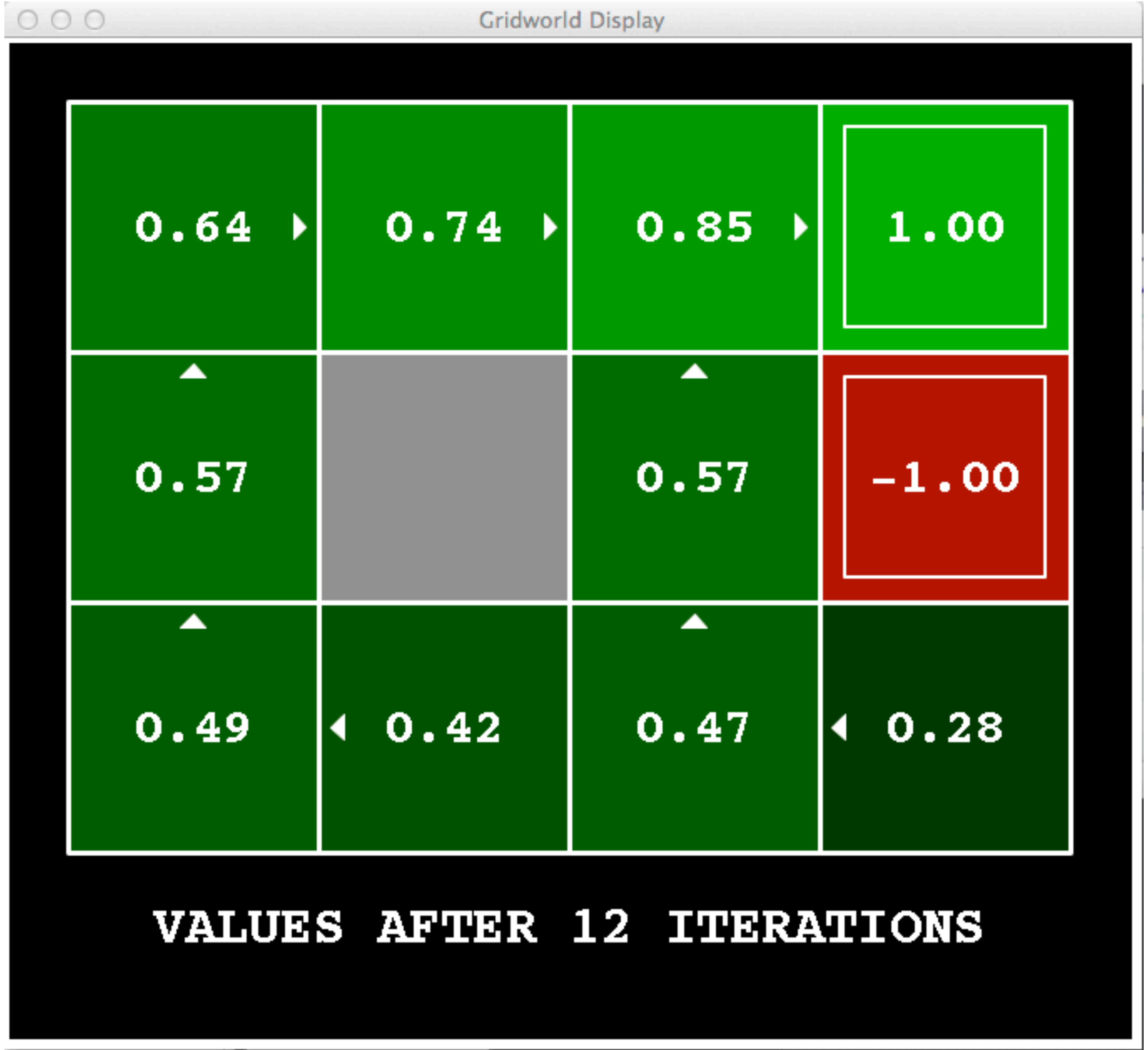
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



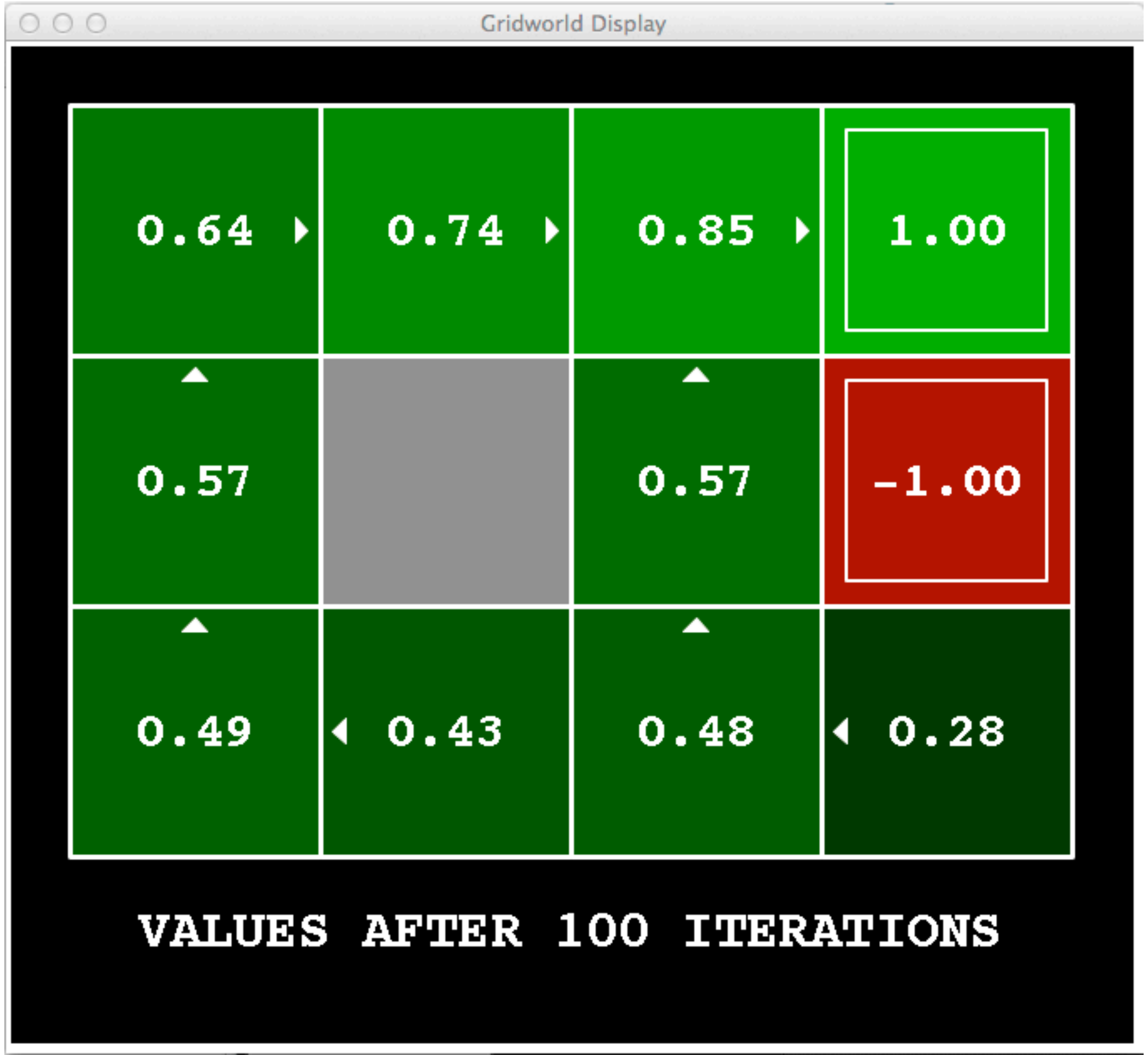
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

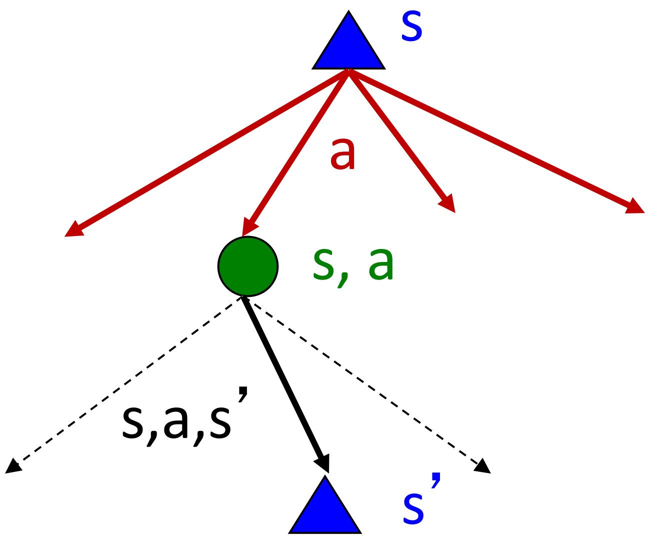


Noise = 0.2
Discount = 0.9
Living reward = 0

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

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, 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 policy using policy extraction

One-step look-ahead:

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

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 (producing optimal values/policies) MDPs

Summary: MDP algorithms

So you want to….

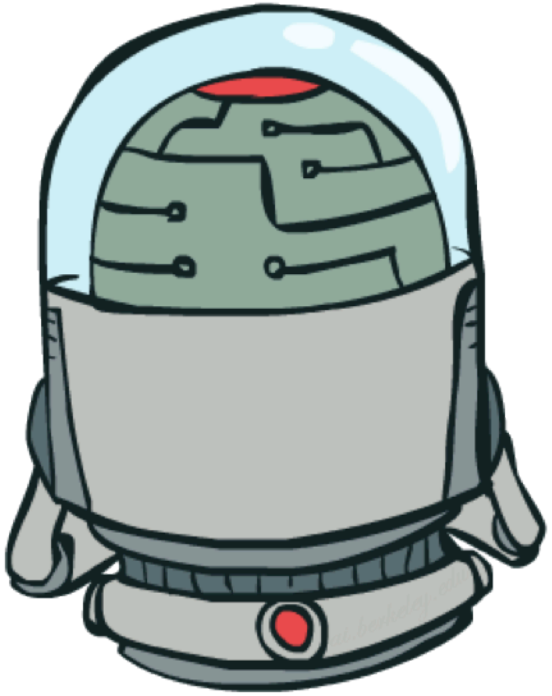
- Compute optimal values: use value iteration or policy iteration
- Compute values for a particular policy: use policy evaluation
- Turn your values into a policy: use policy extraction (one-step lookahead)

Hey, 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

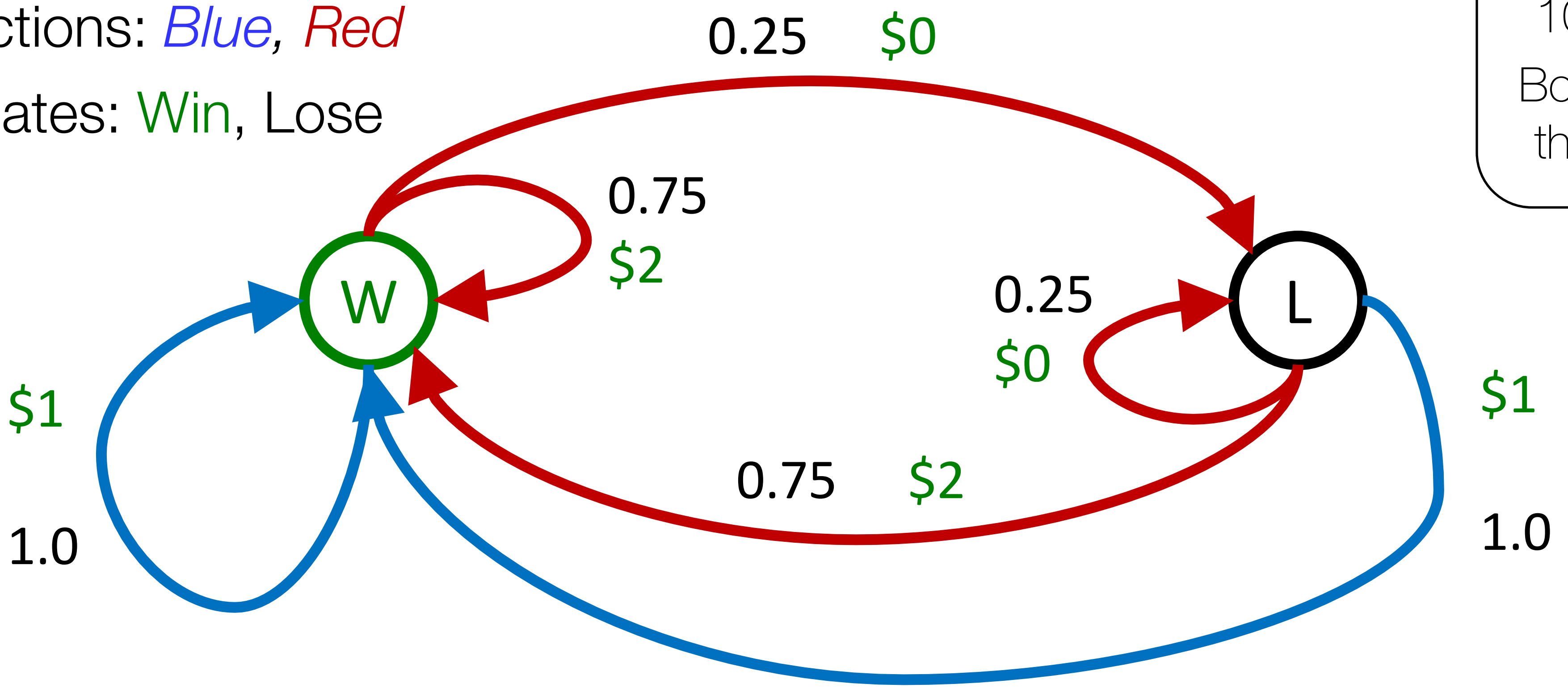
From MDPs to *reinforcement learning*…

Double bandits



Double bandits

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



No discount
100 time steps
Both states have the same value

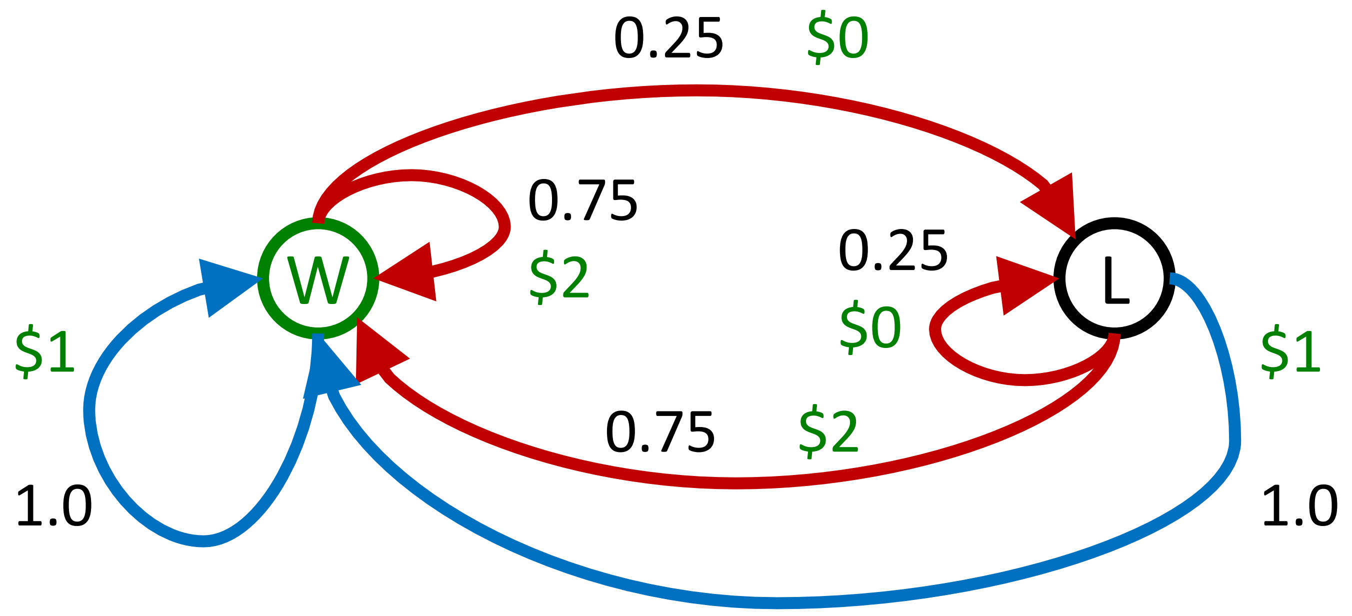
Offline planning

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



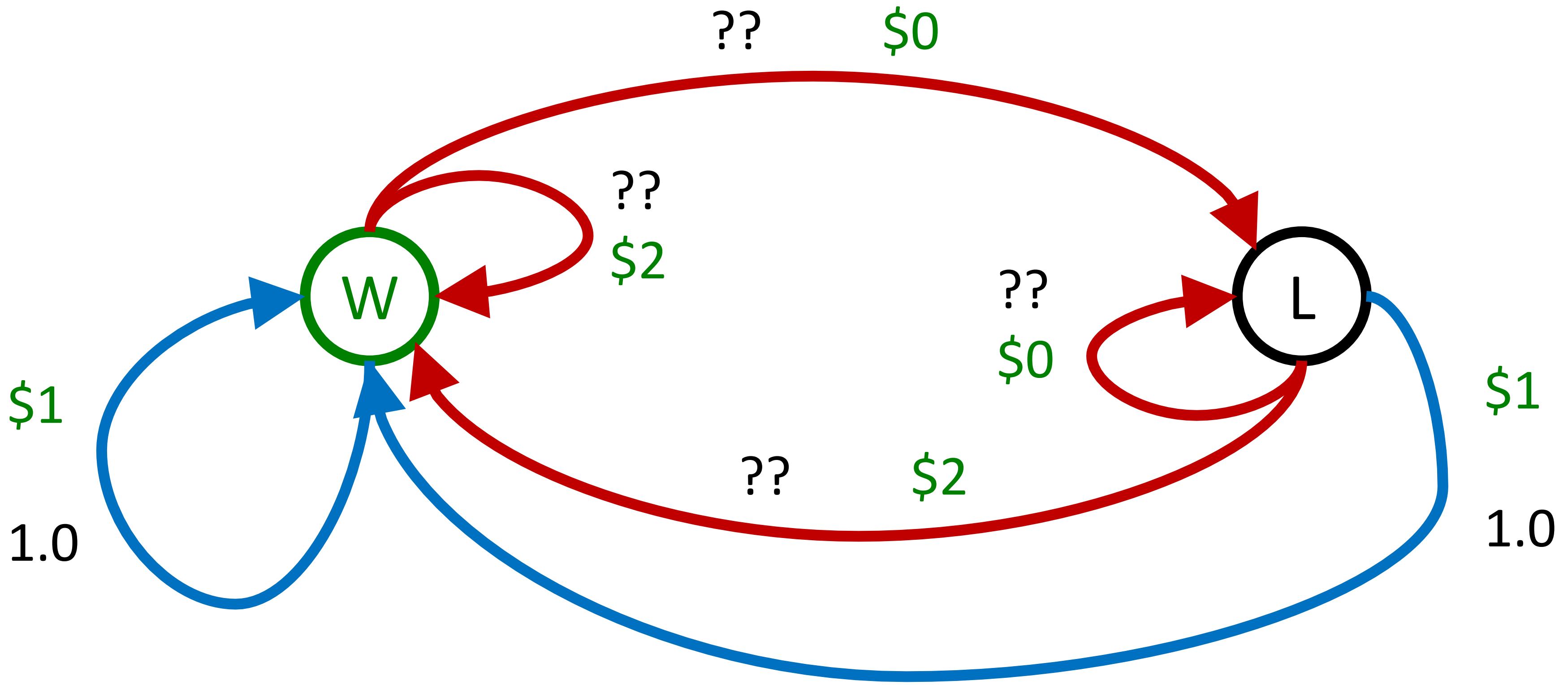
Let's play!

$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*

Remember: no class Tuesday -- work on HWs!