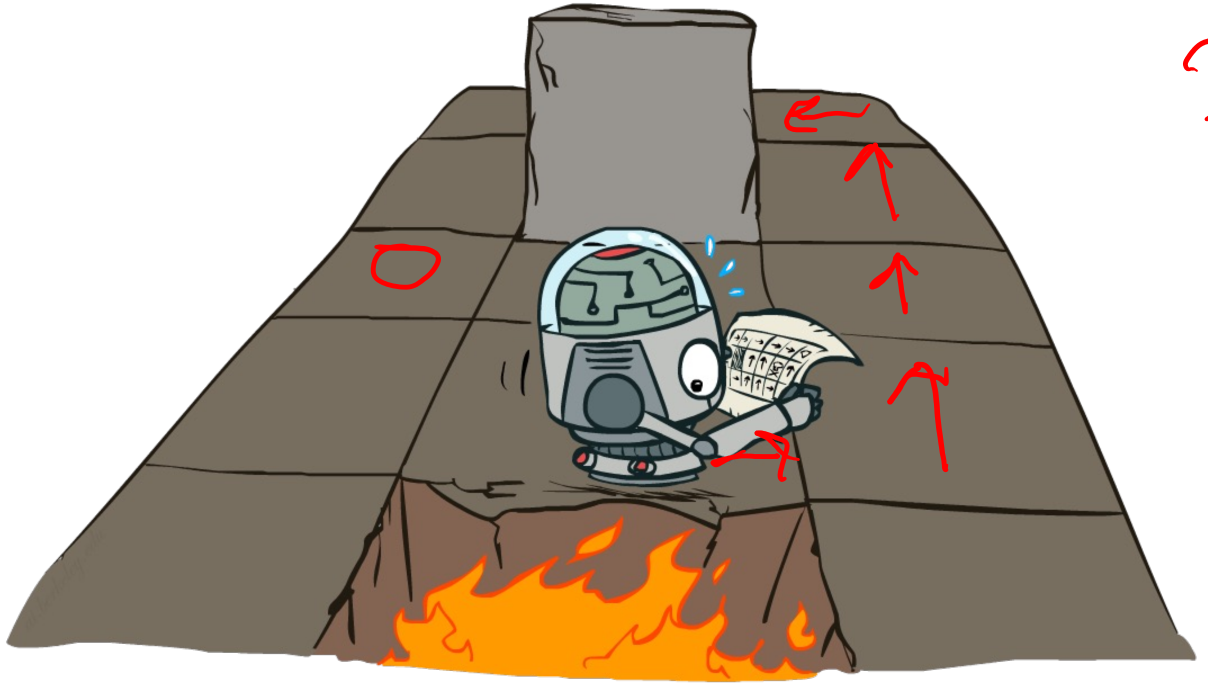
AI: Representation and Problem Solving

Markov Decision Processes II

$\pi(s) \to a$

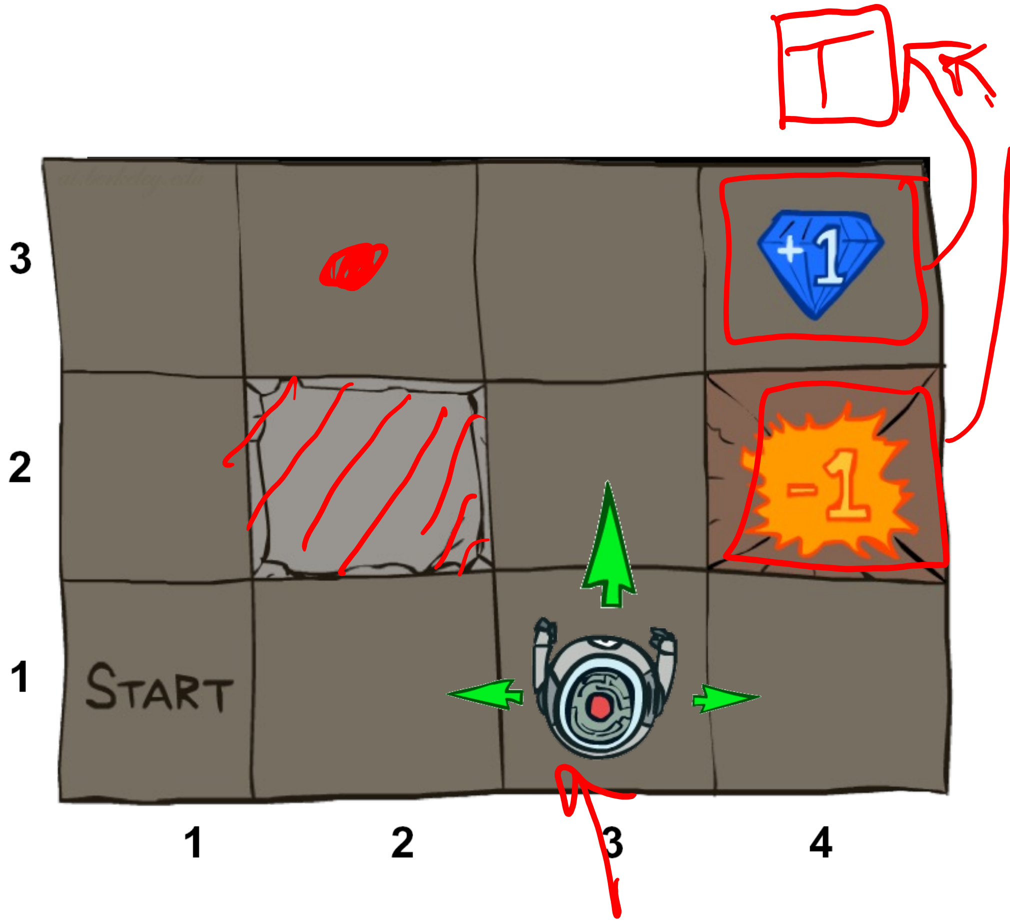


Instructors: Tuomas Sandholm and Vincent Conitzer

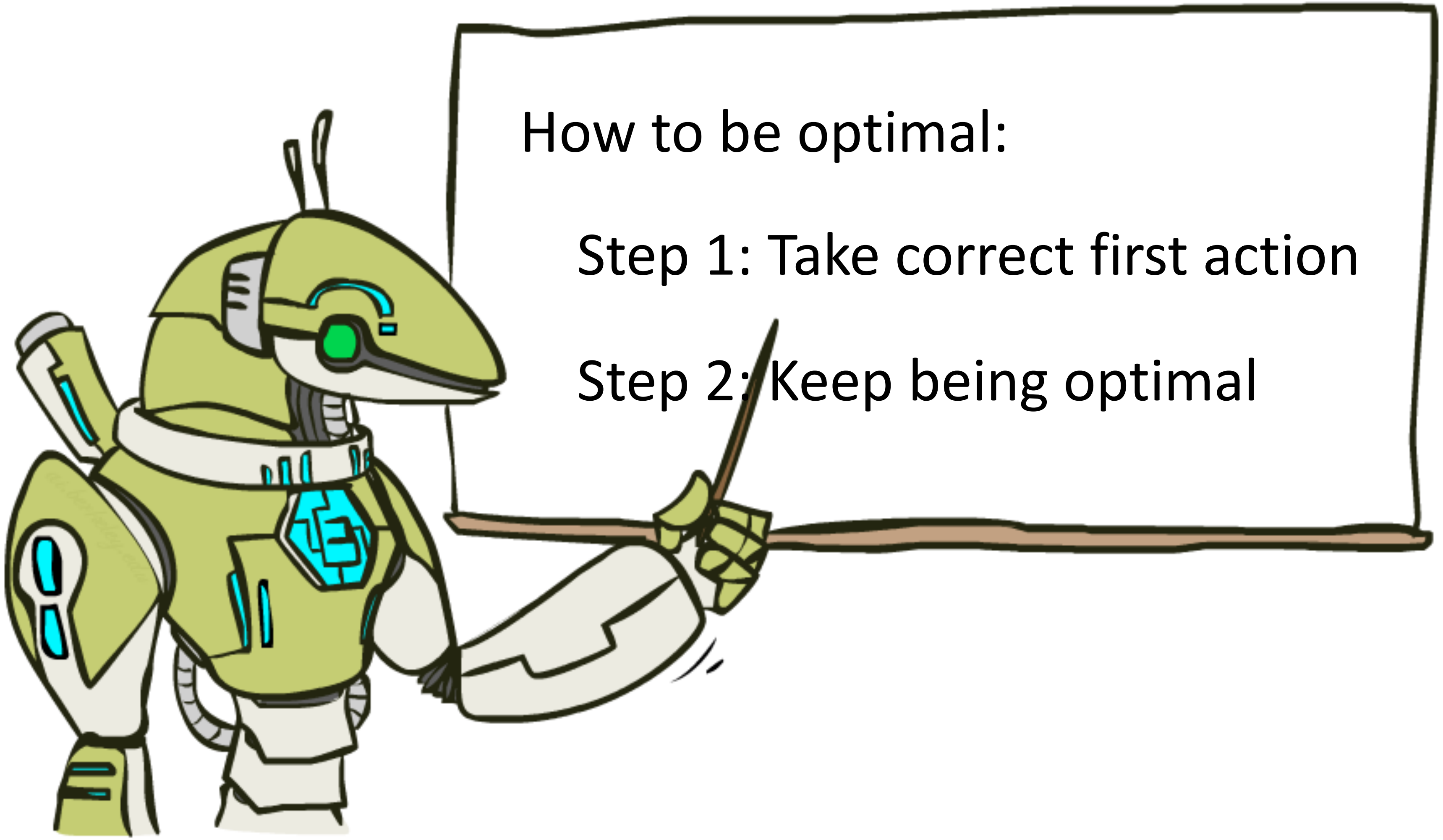
Slide credits: CMU AI and http://ai.berkeley.edu

Recap: 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)
- In the previous lecture we showed an algorithm for solving MDPs: **value iteration**

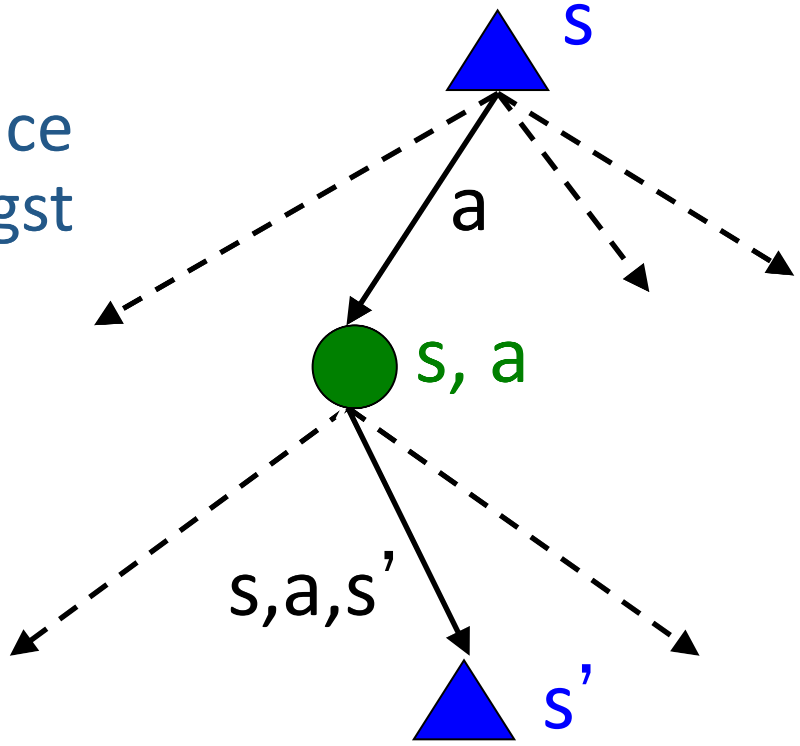


The Bellman Equations



The Bellman Equations

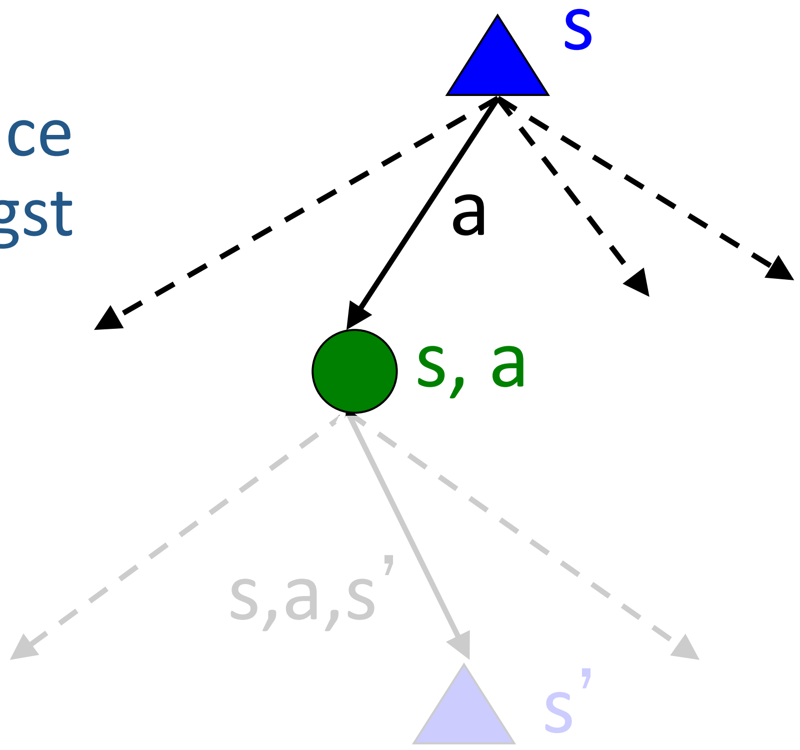
Definition of “optimal utility” via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values



The Bellman Equations

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

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

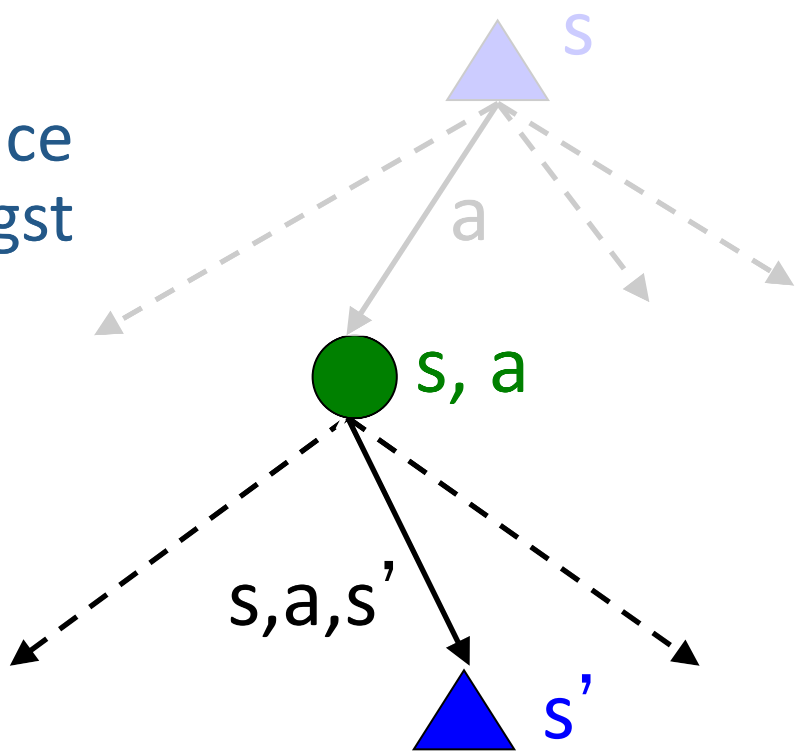


The Bellman Equations

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

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

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



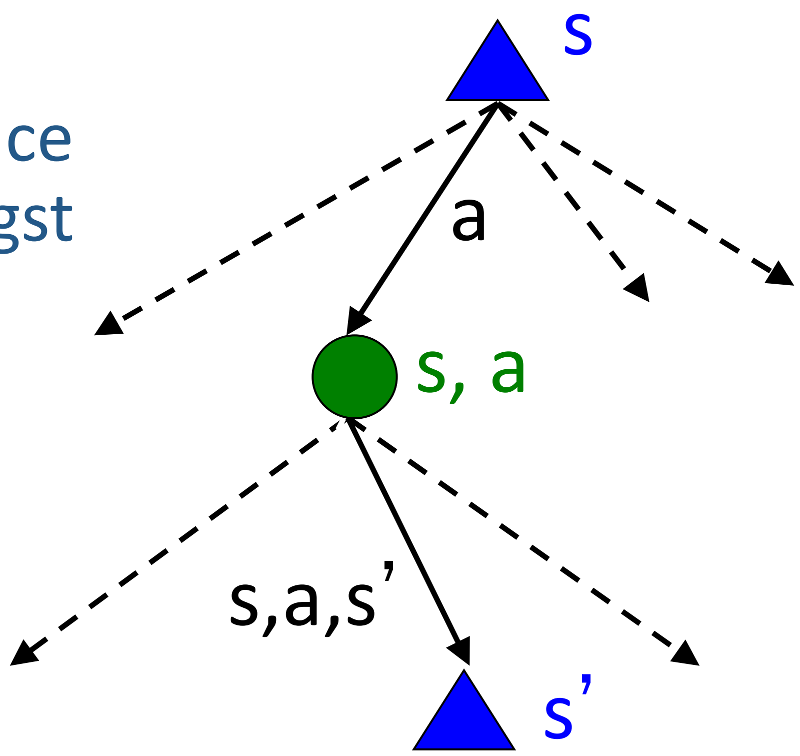
The Bellman Equations

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

MDP Notation

Standard expectimax: $V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$

Bellman equations: $V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$ $\forall s$

Value iteration: $V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \quad \forall s$

Value Iteration

Bellman equations characterize the optimal values:

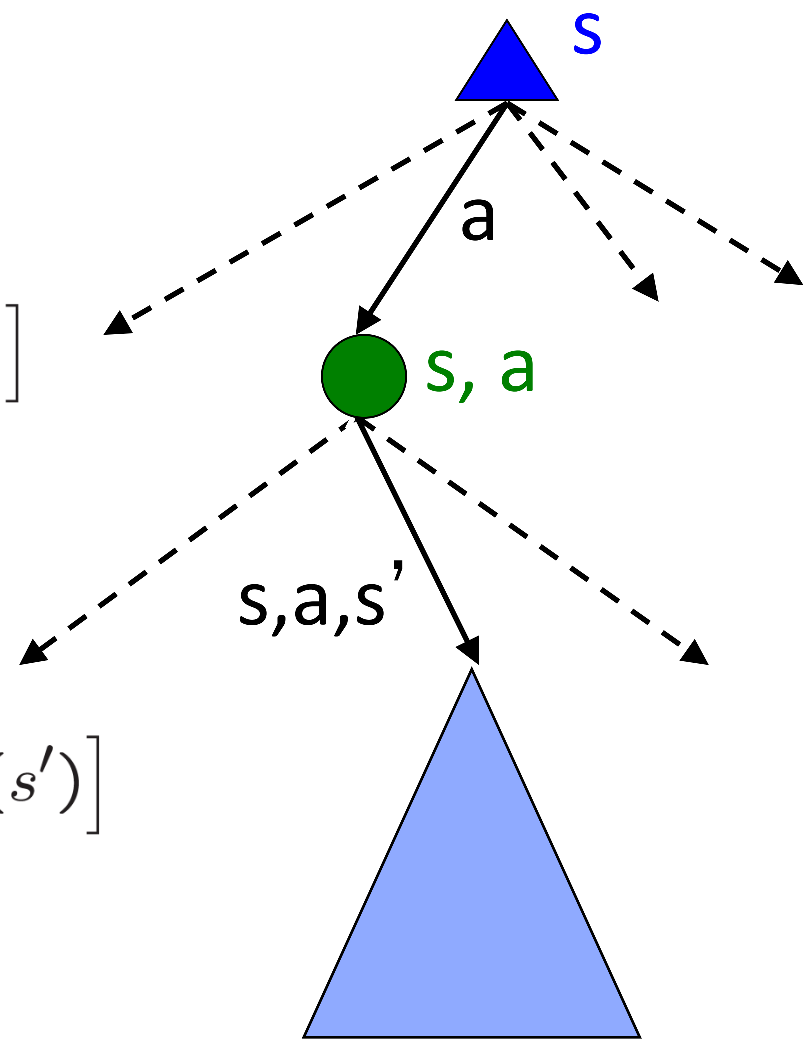
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

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 a fixed point solution method

- … though the V_k vectors are also interpretable as time-limited values

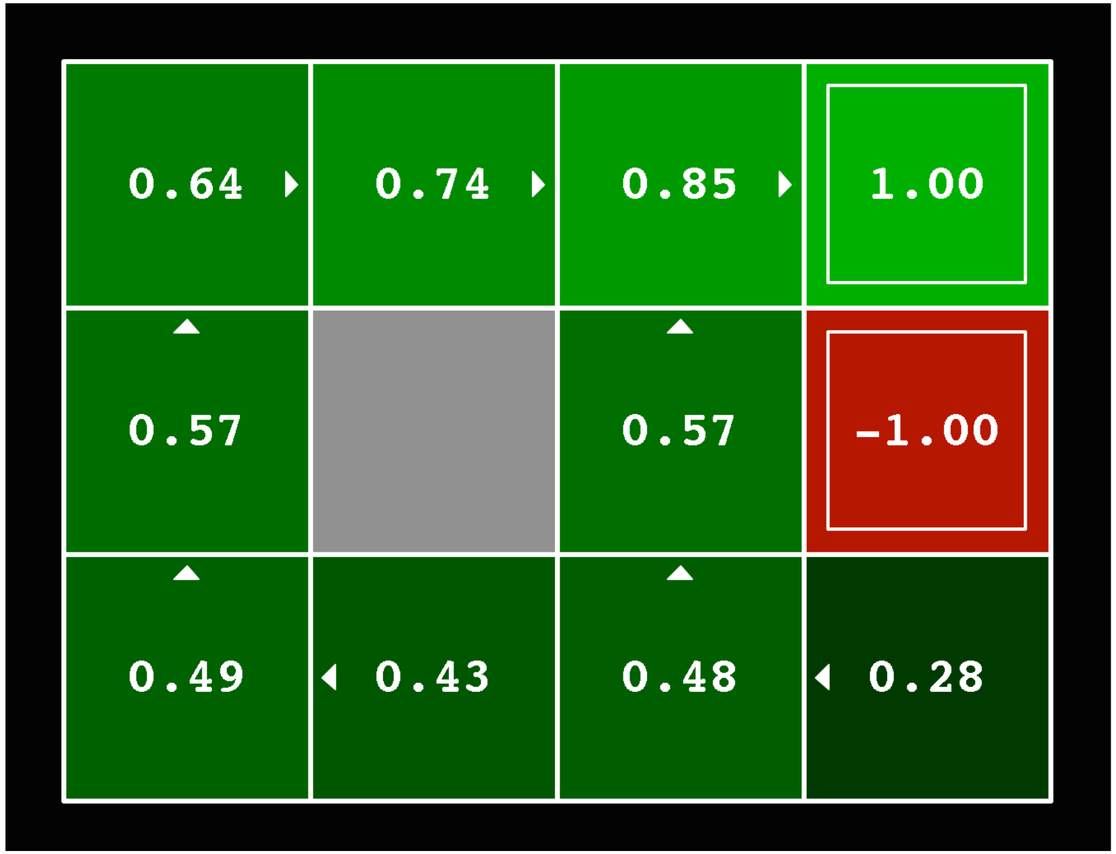


Solved MDP! Now what?

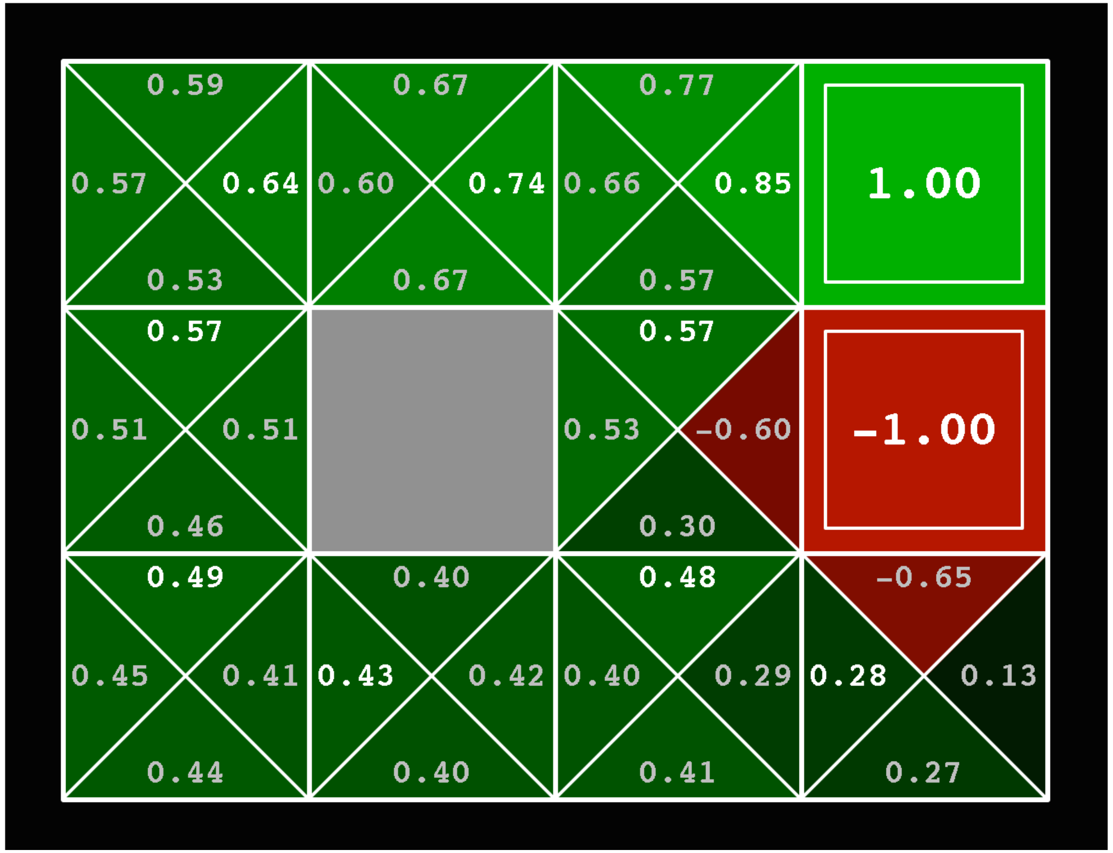
$\pi(s) \rightarrow a$

What are we going to do with these values??

$V^*(s)$



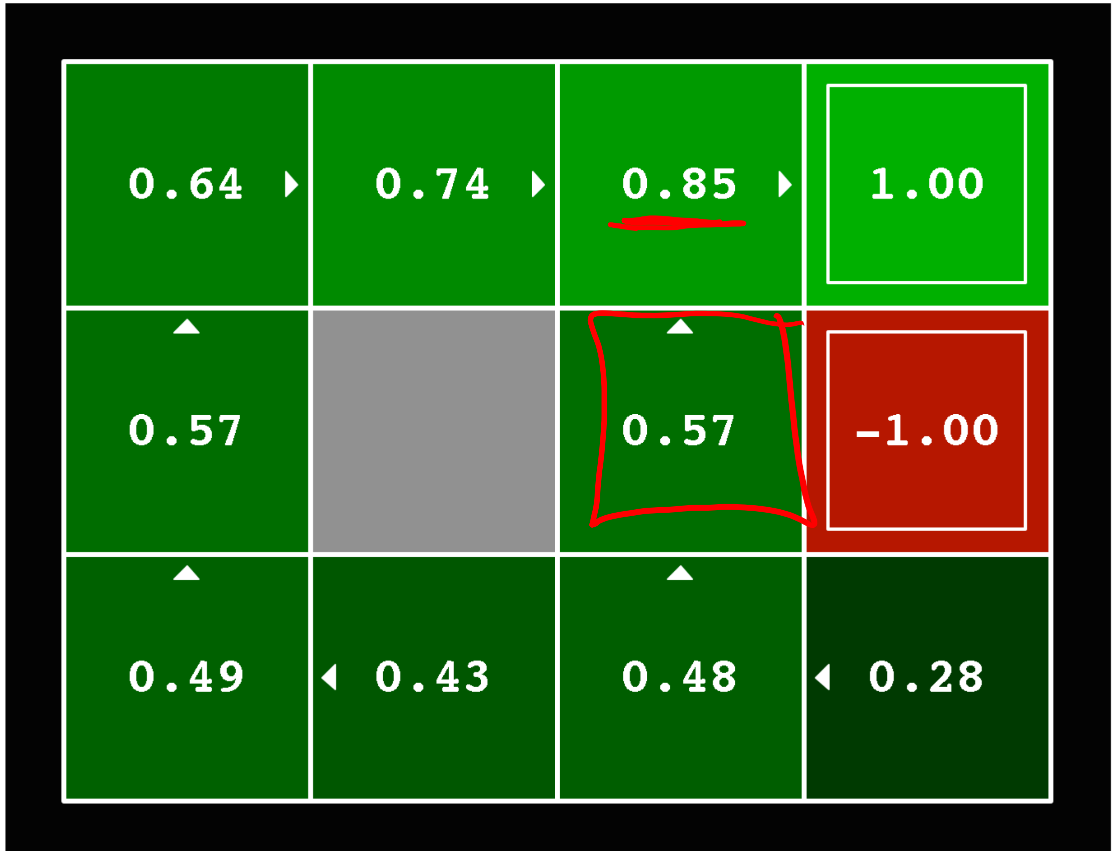
$Q^*(s,a)$

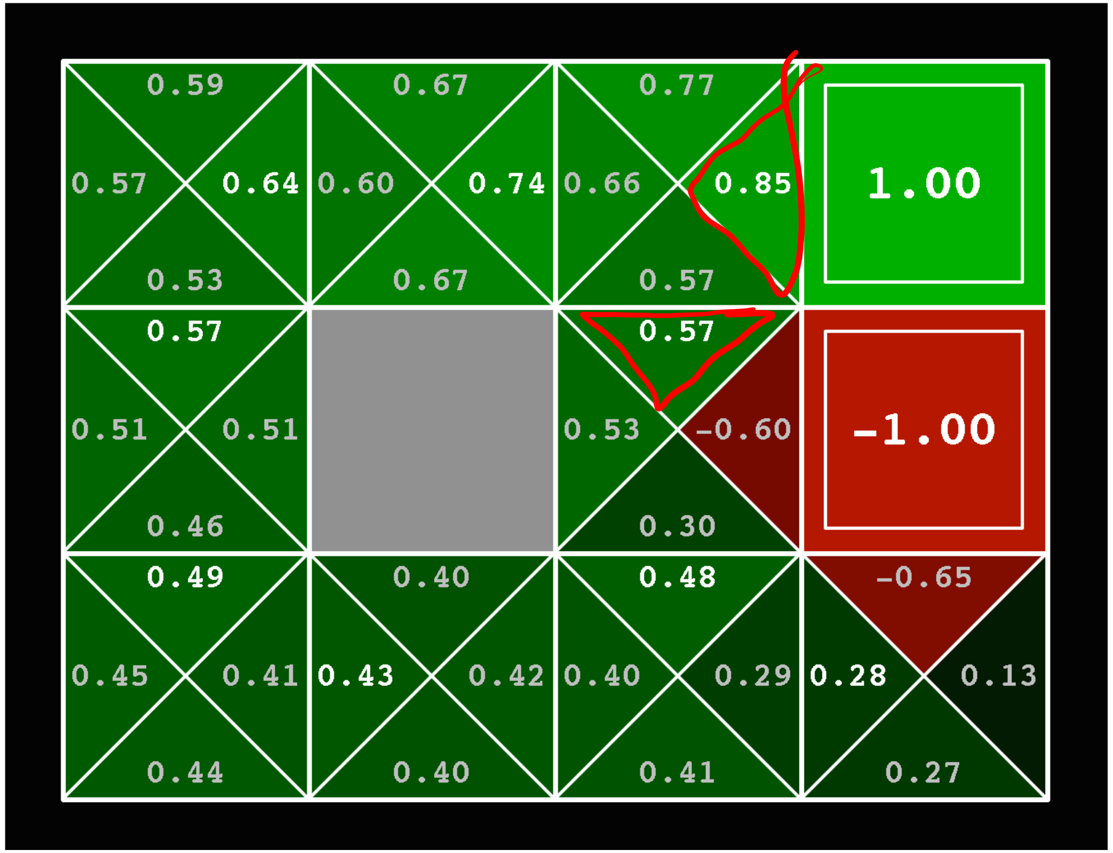


Poll

If you need to extract a policy, would you rather have

A) Values, B) Q-values?

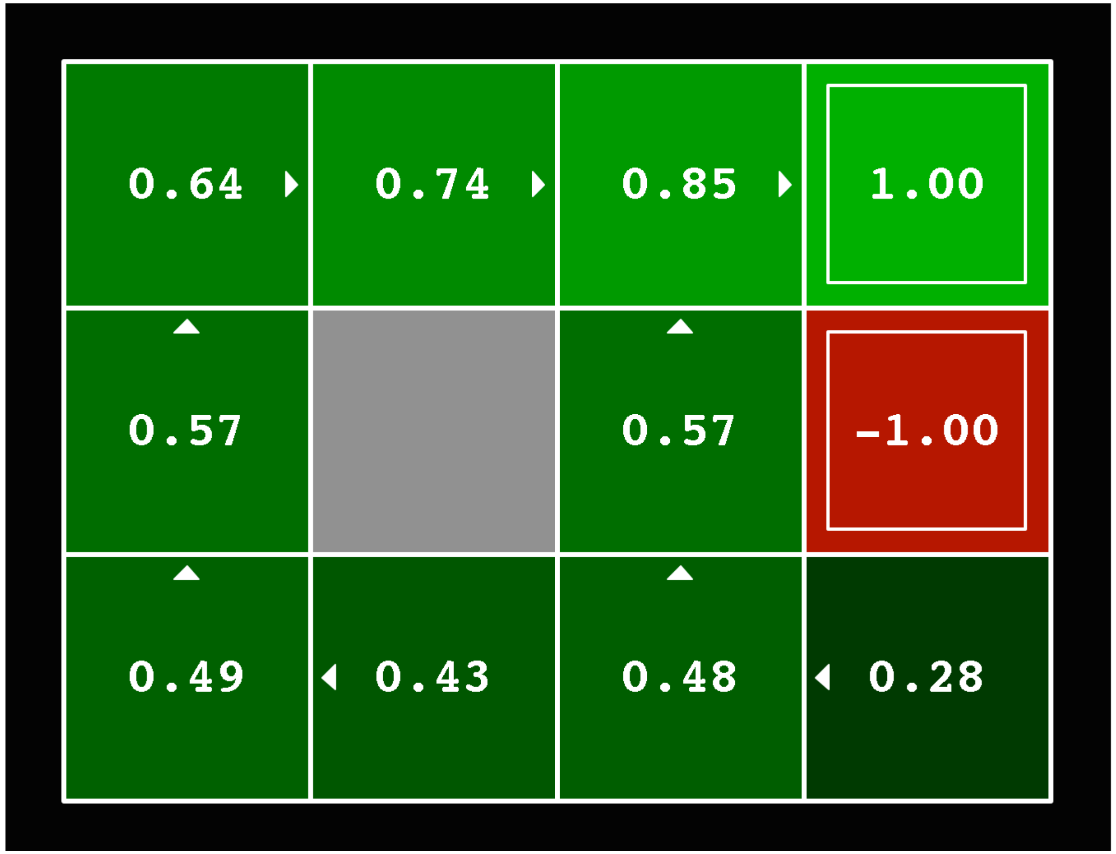


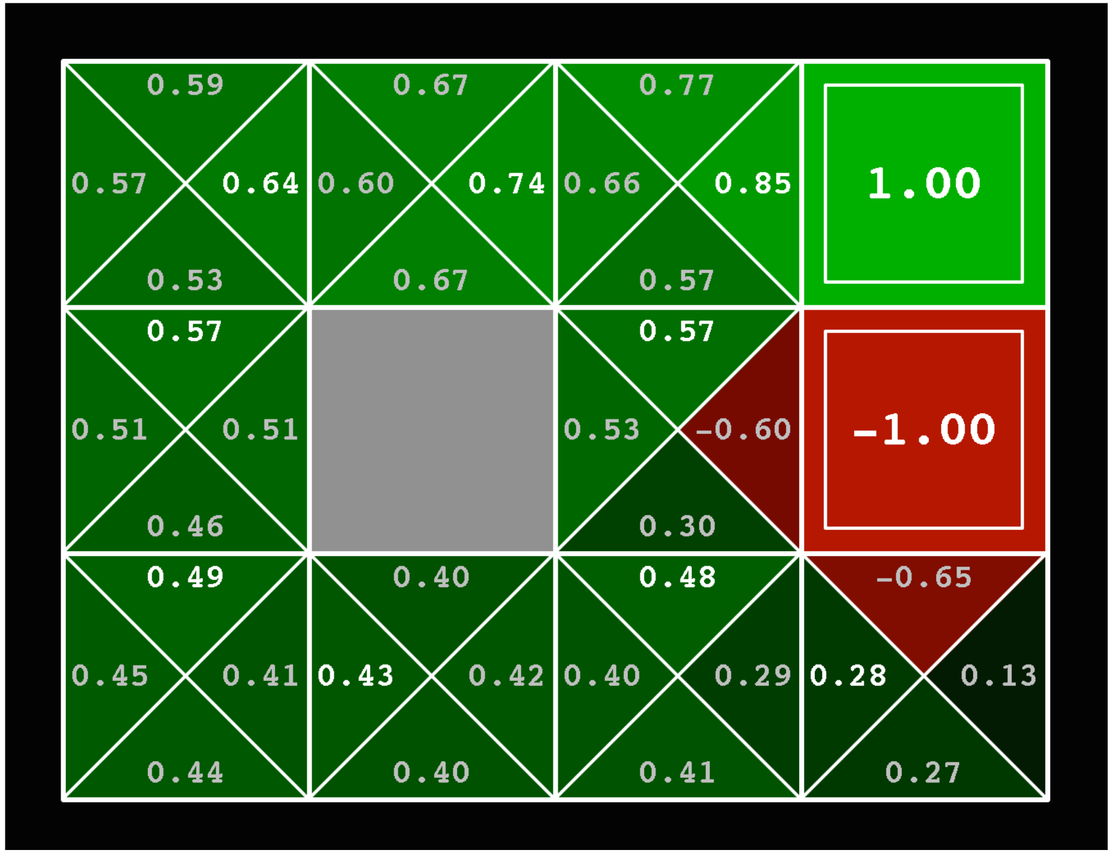


Poll

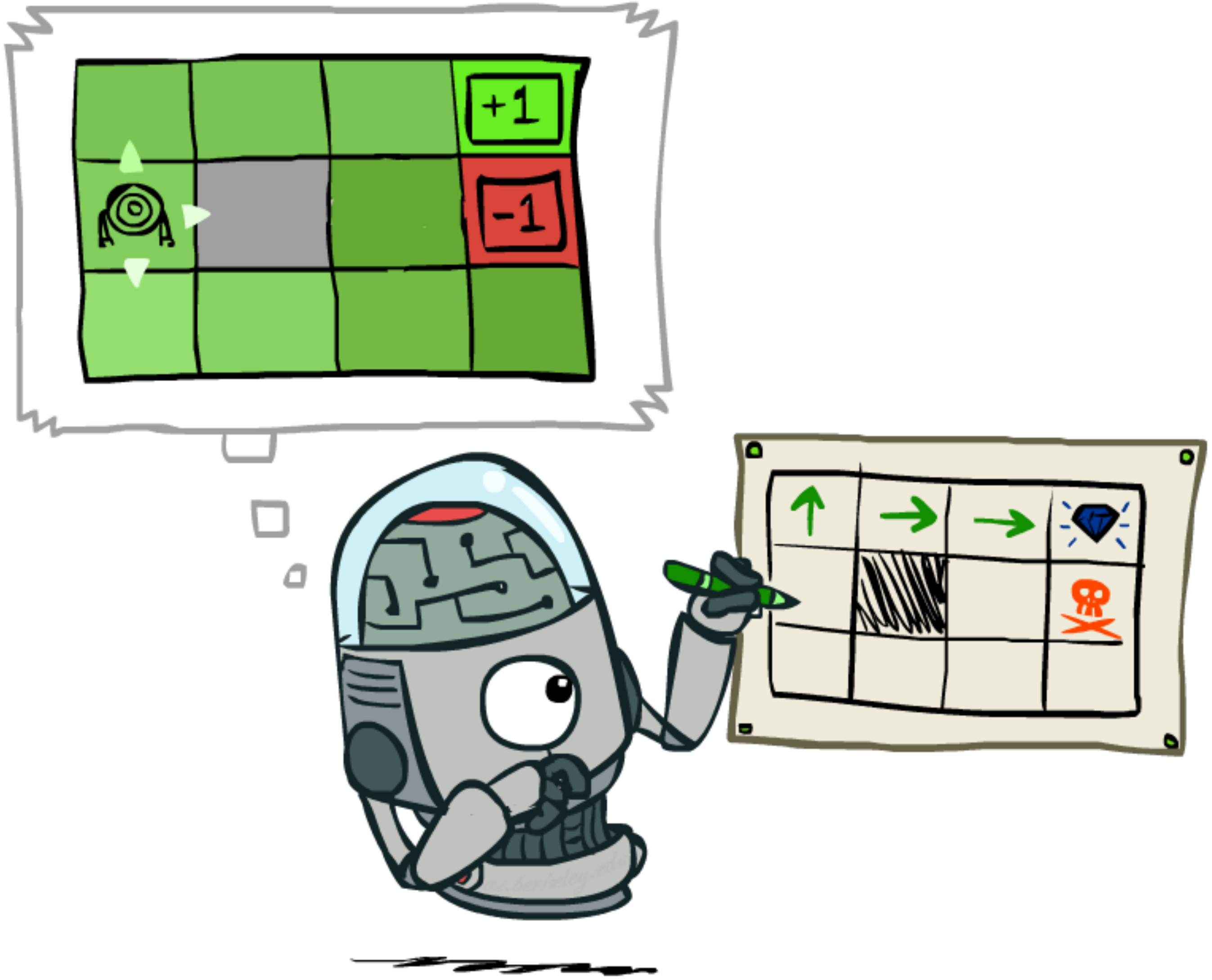
If you need to extract a policy, would you rather have

A) Values, B) Q-values ?





Policy Extraction



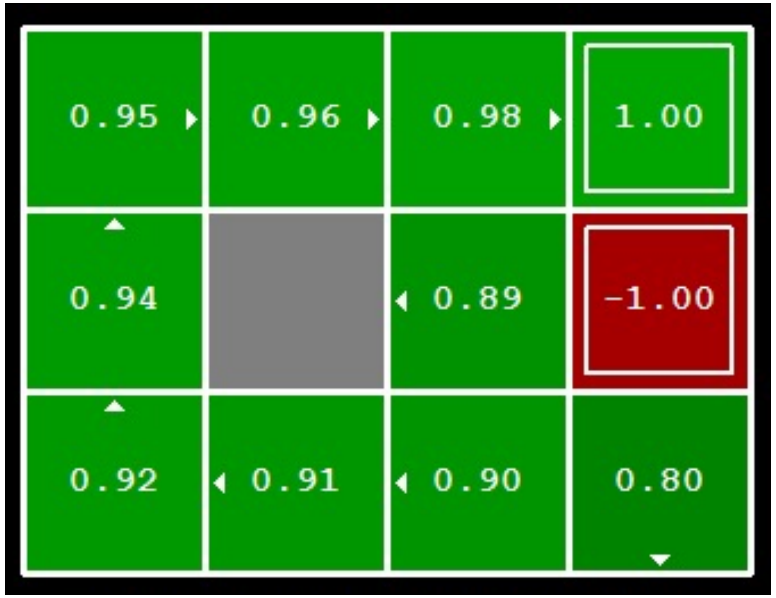
Computing Actions from Values

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

How should we act?

- It's not obvious!

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

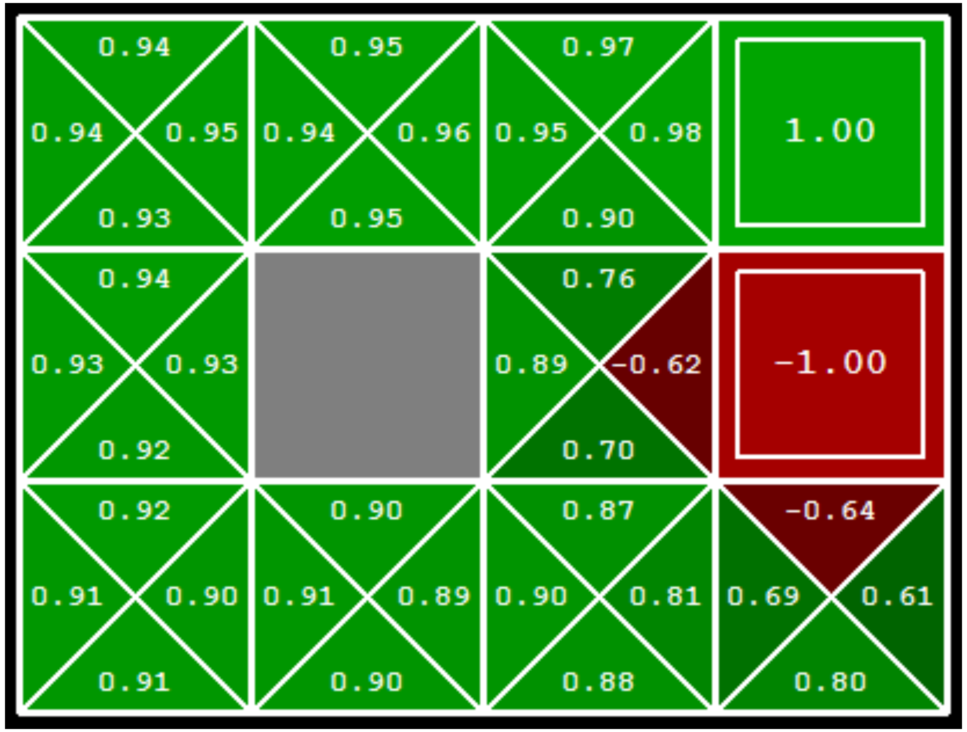


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

Computing Actions from Q-Values

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



How should we act?

- Completely trivial to decide!

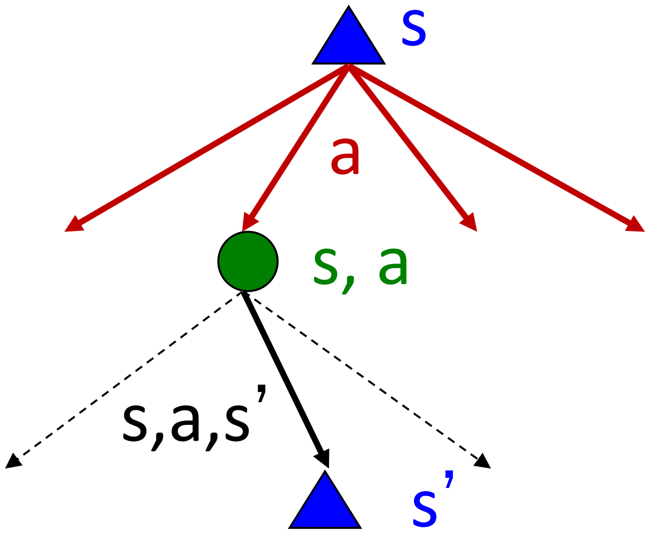
$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

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

Value Iteration Notes

Value iteration repeats the Bellman updates:

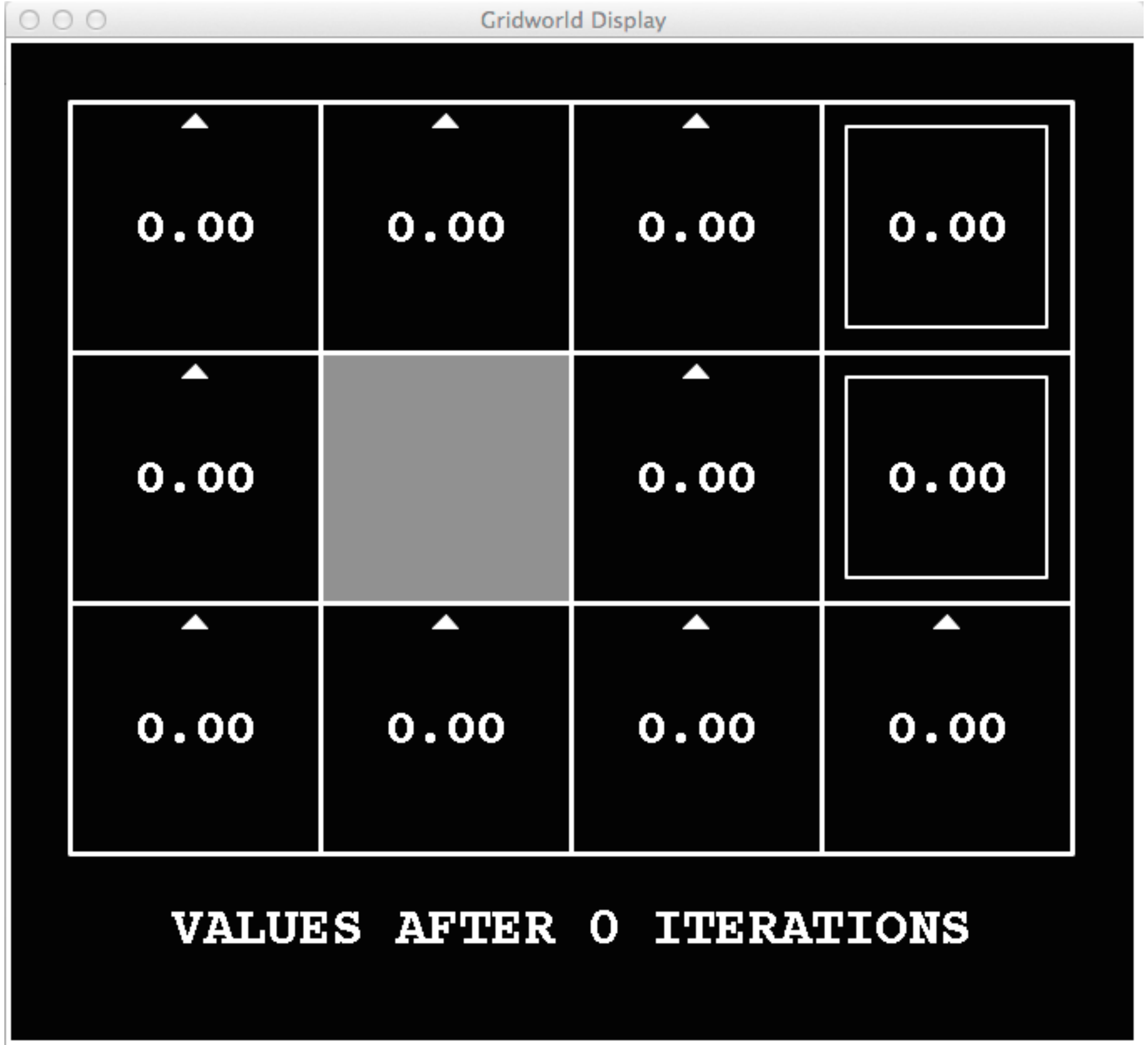
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$



Things to notice when running value iteration:

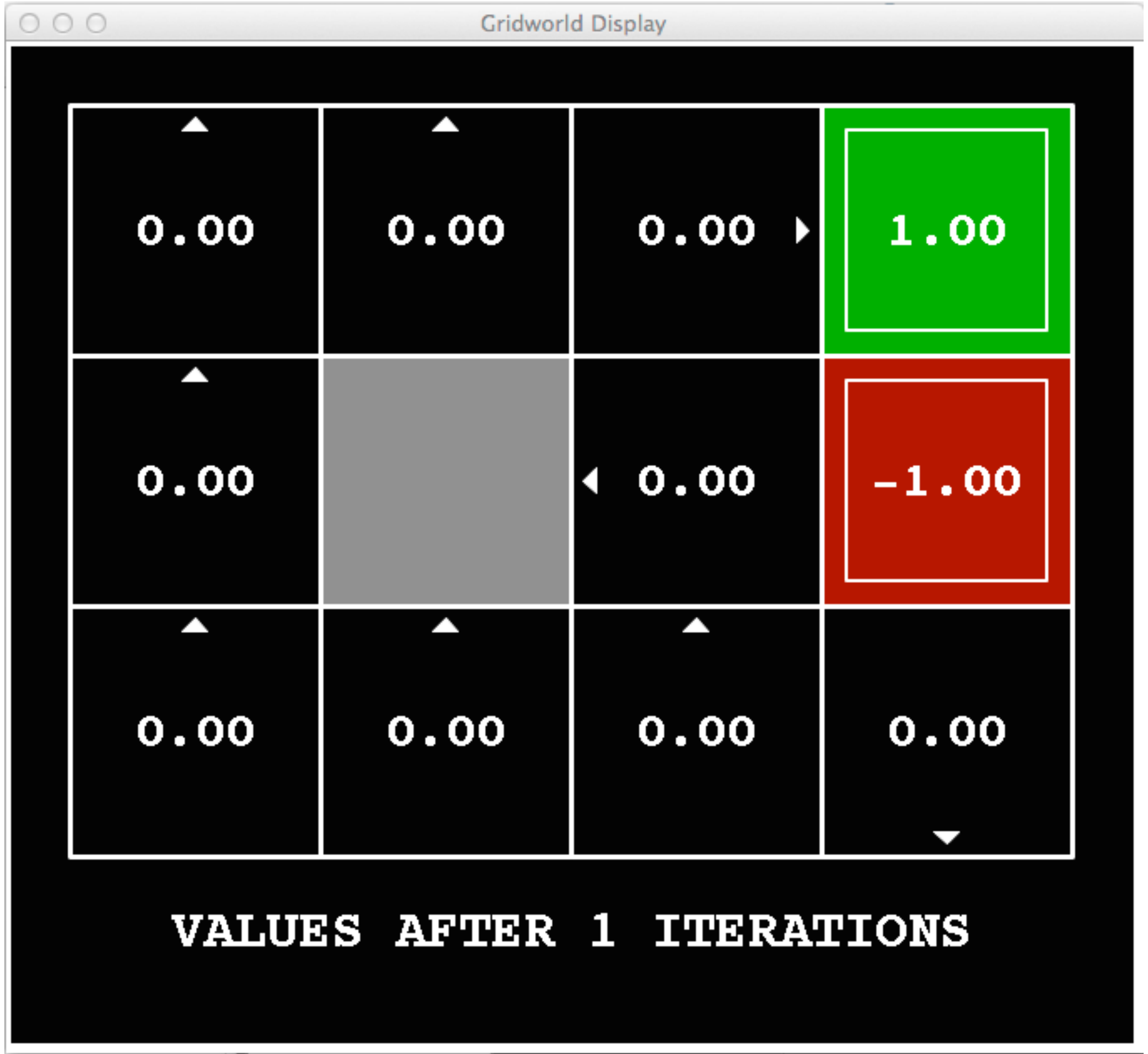
- It's slow – O(S^2A) per iteration
- The "max" at each state rarely changes
- The optimal policy appears before the values converge (but we don't know that the policy is optimal until the values converge)

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

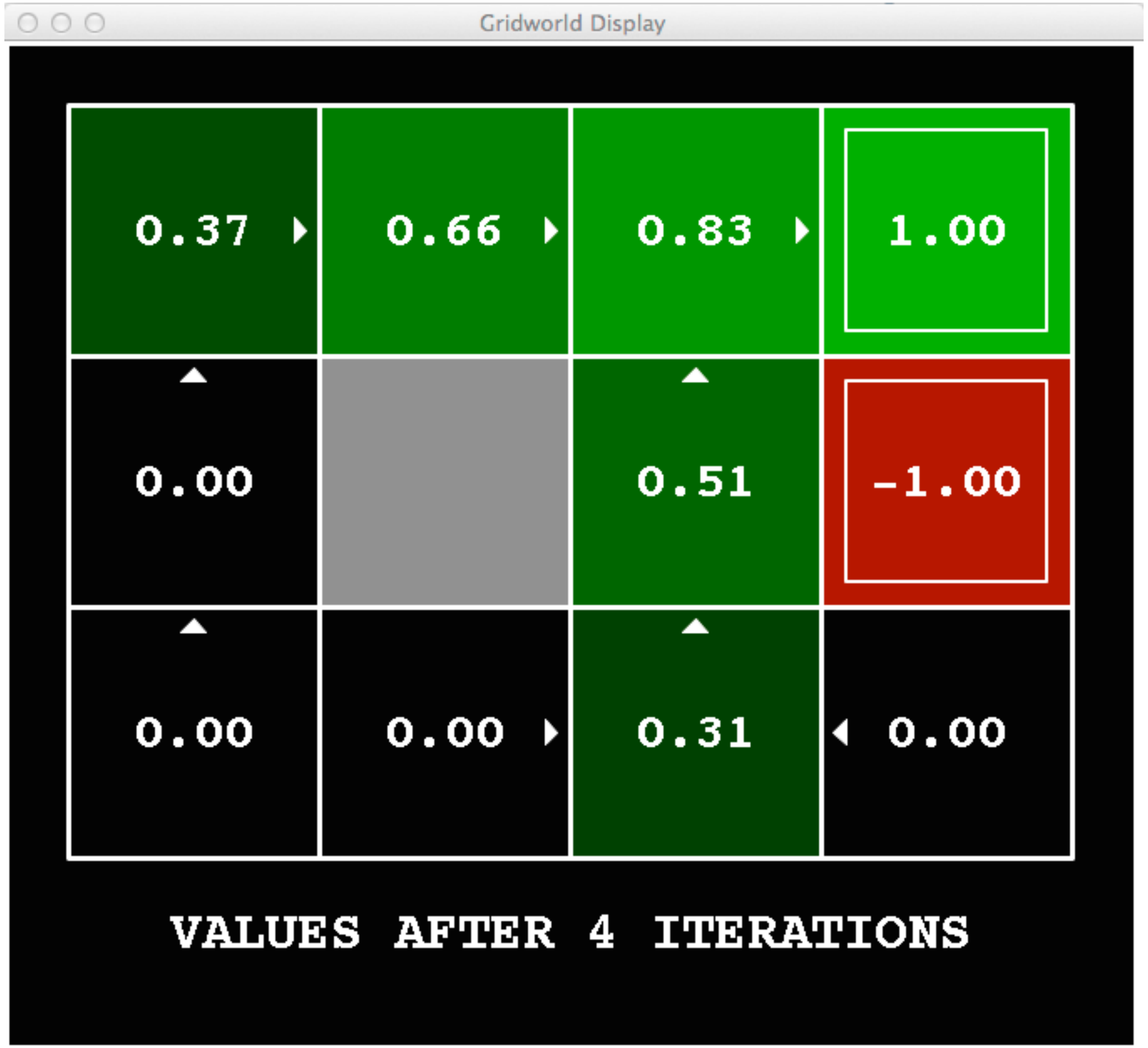
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

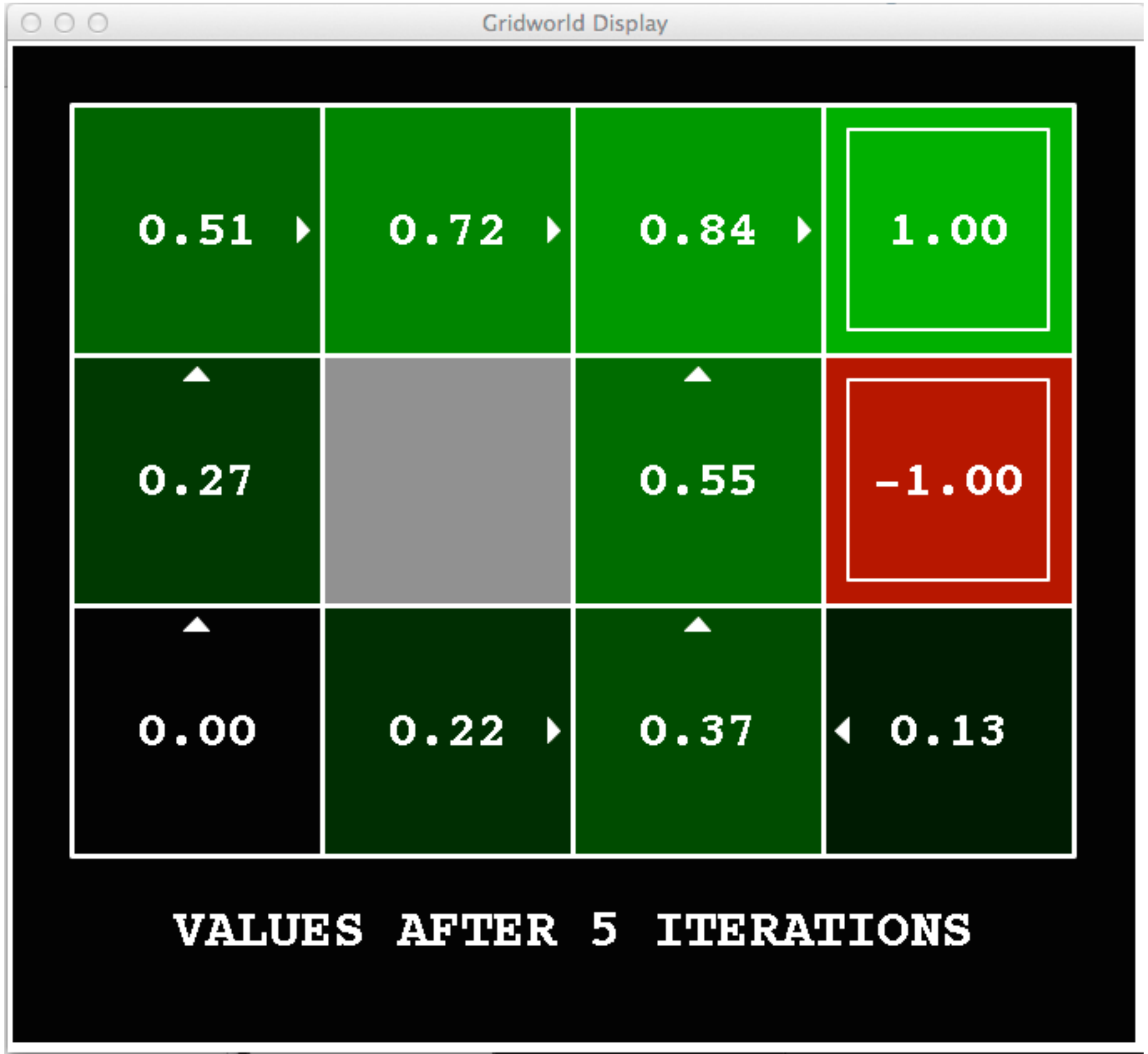
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



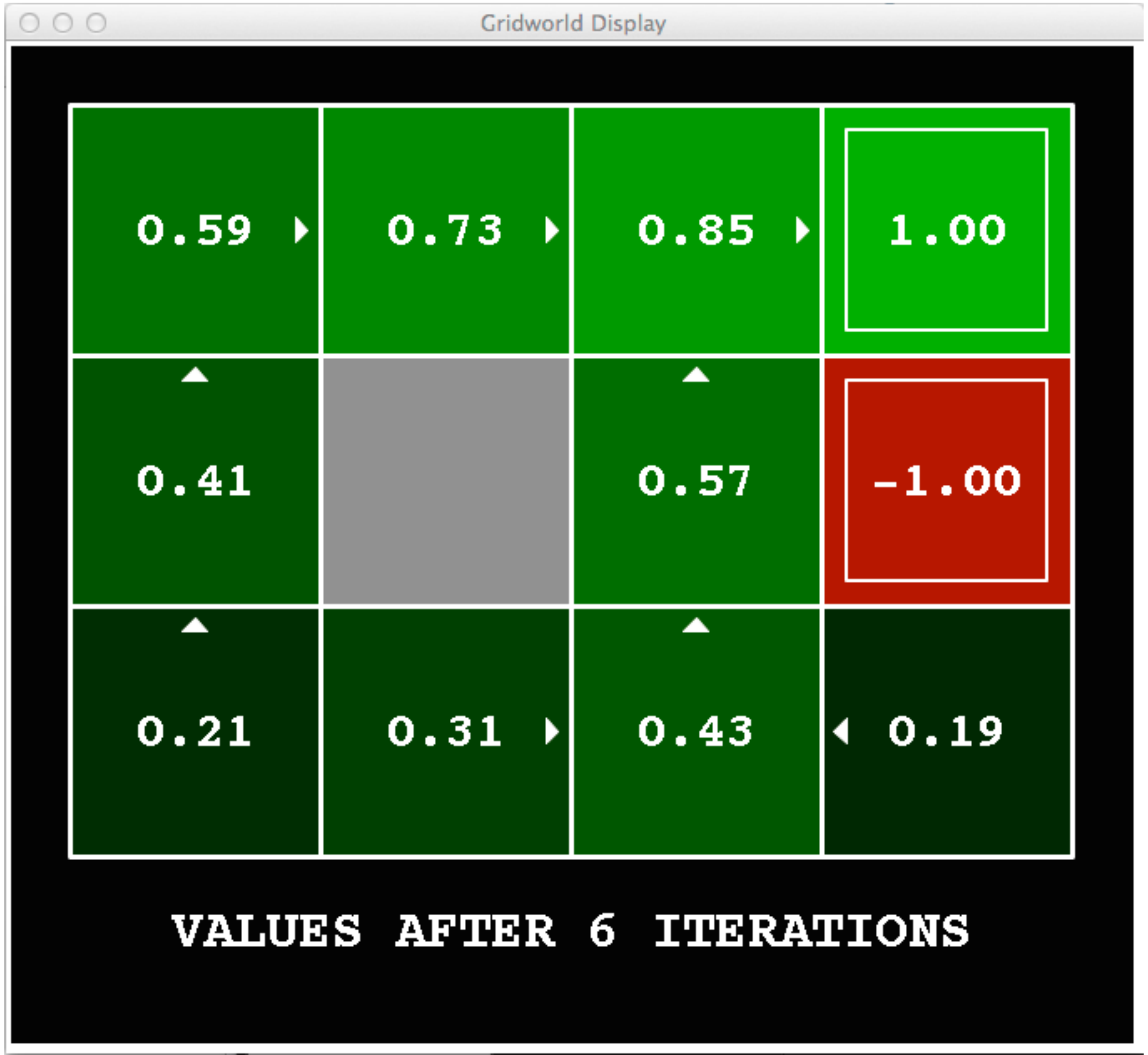
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



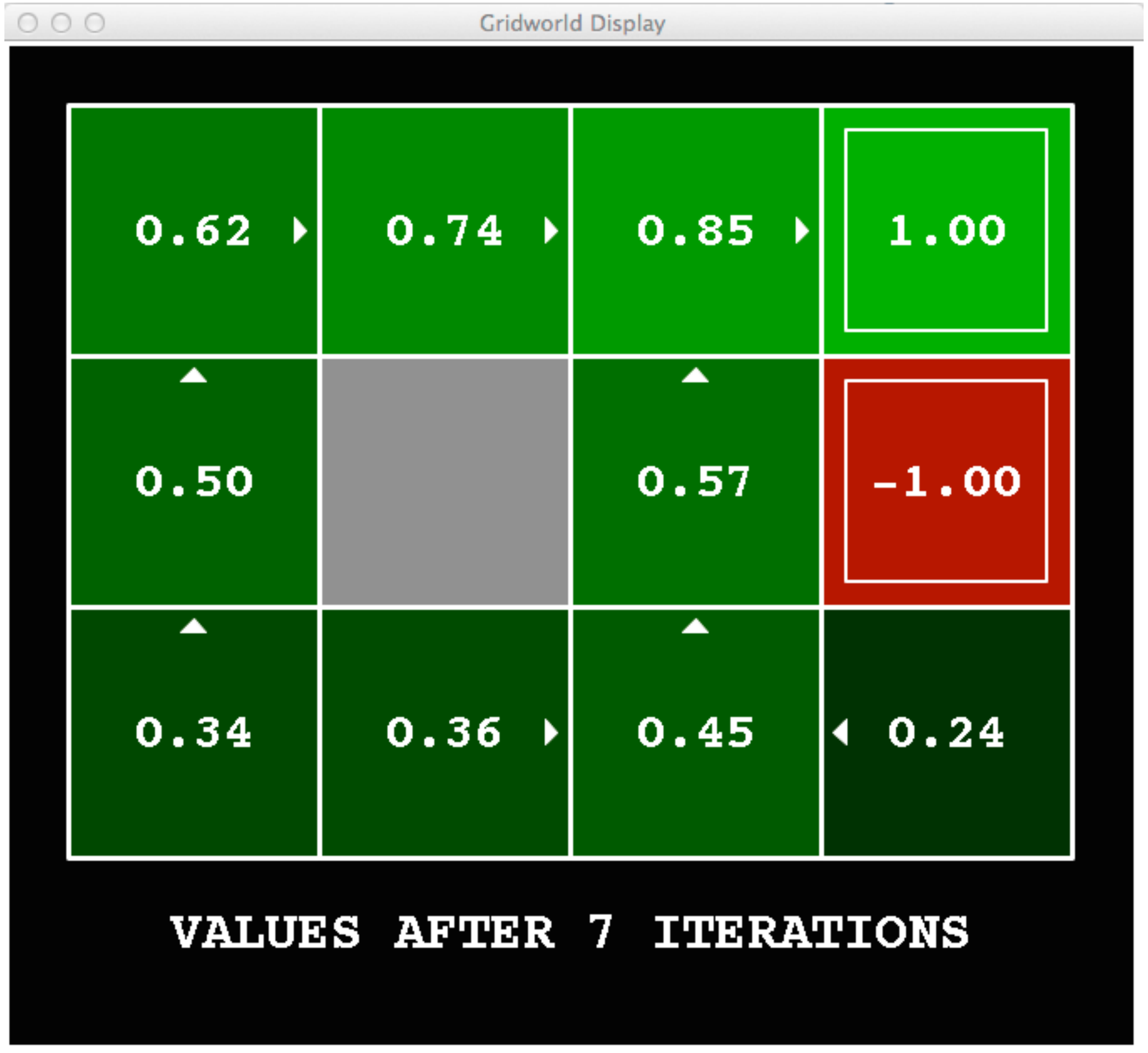
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



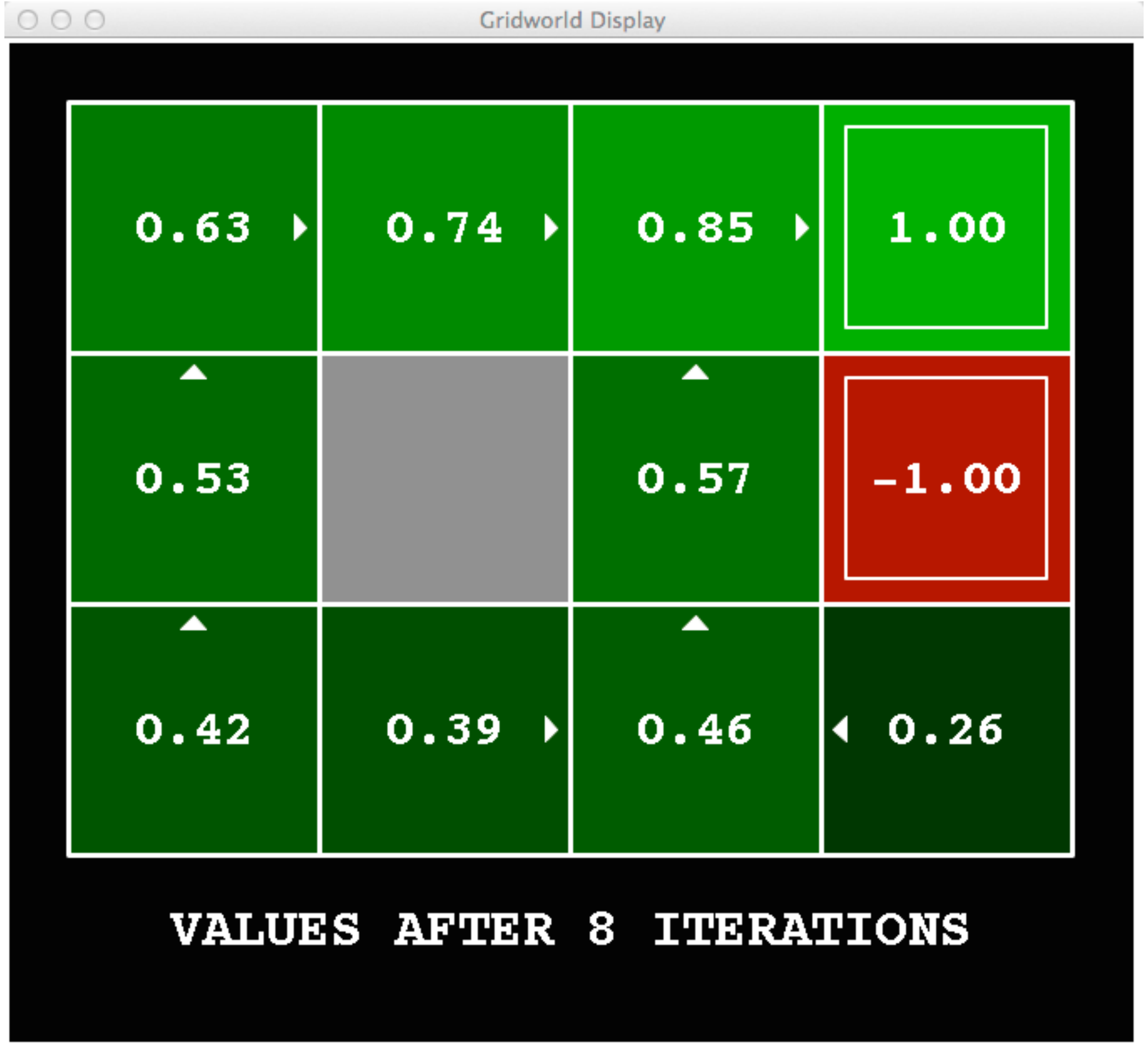
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



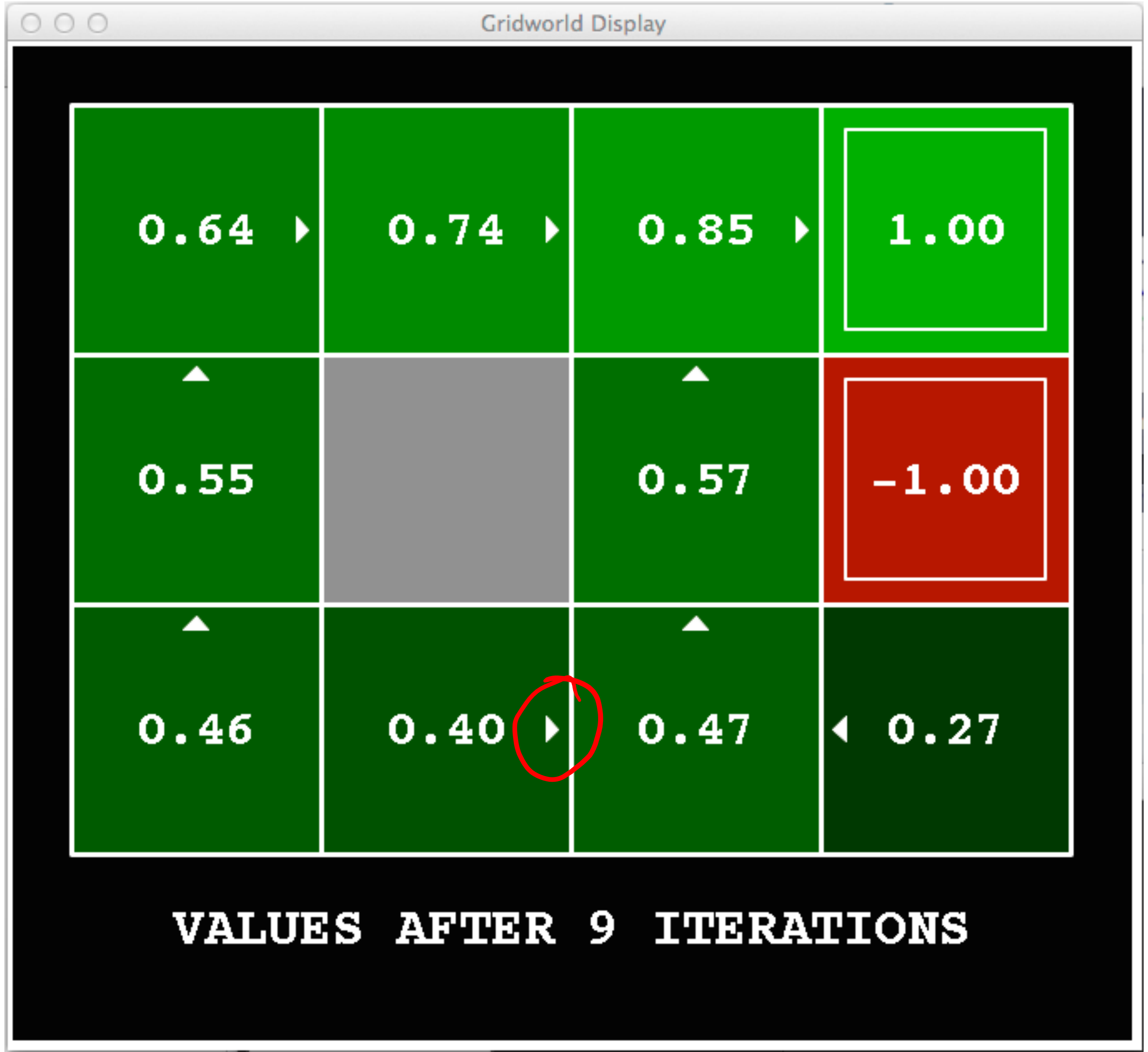
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



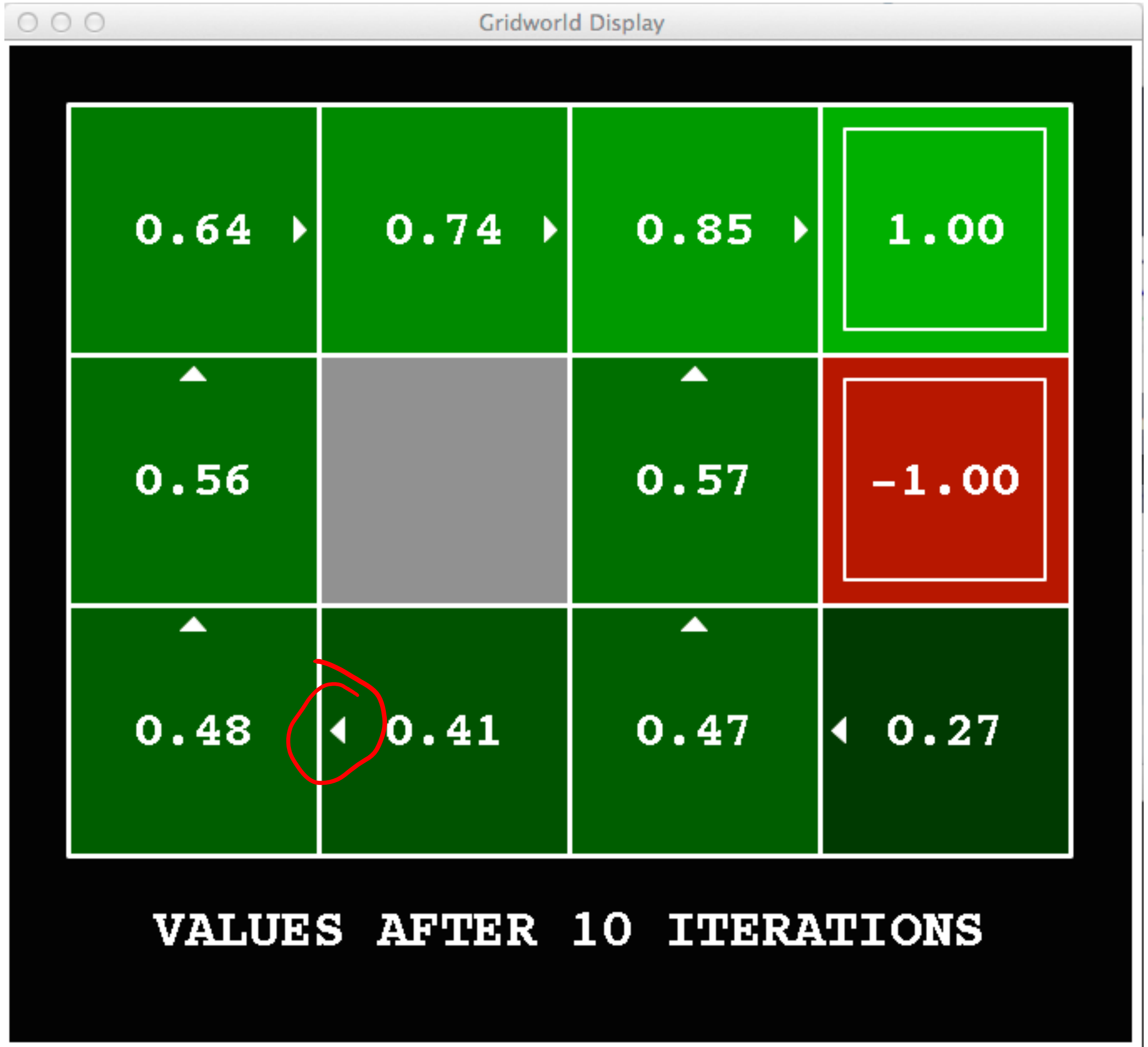
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



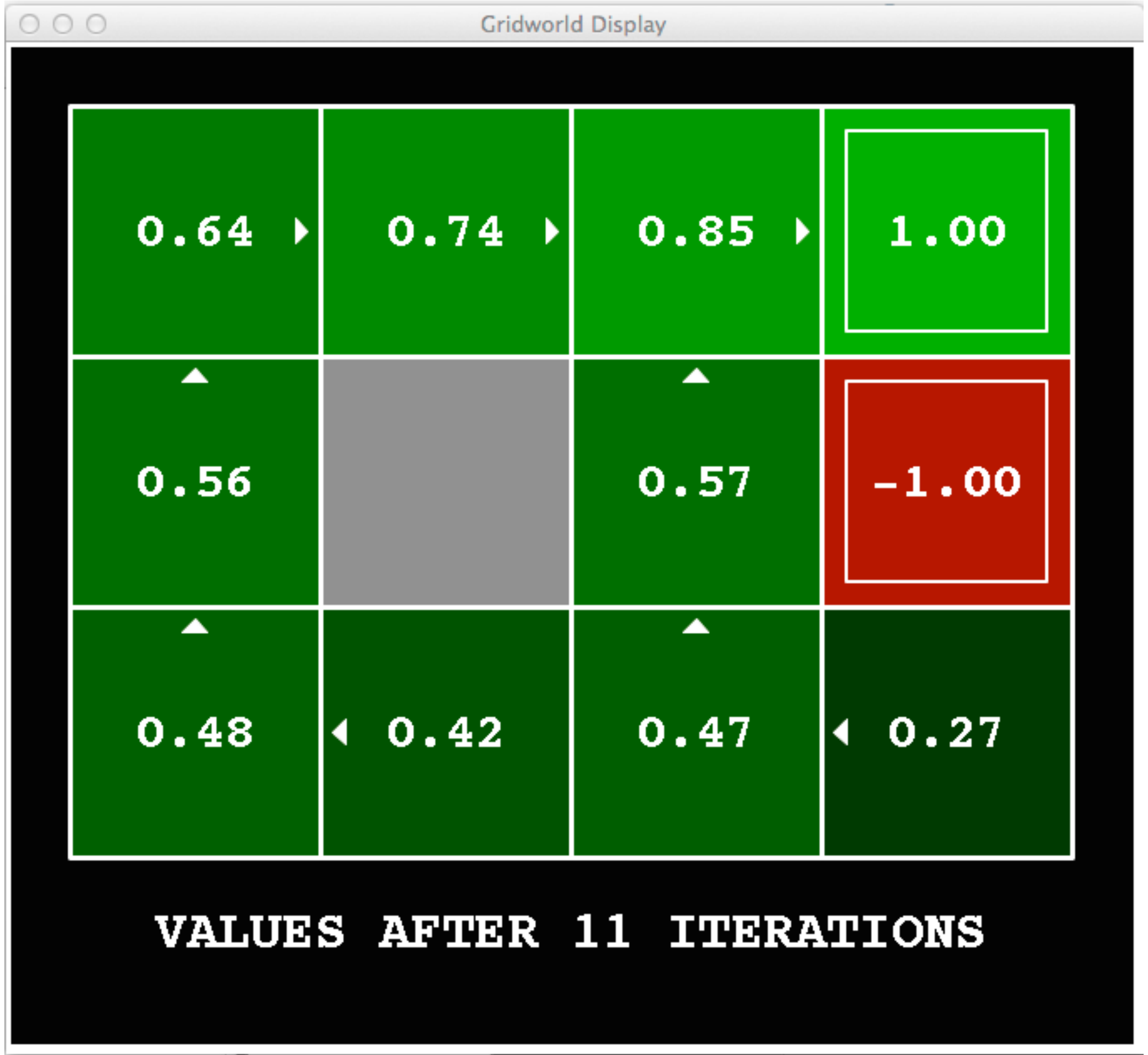
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



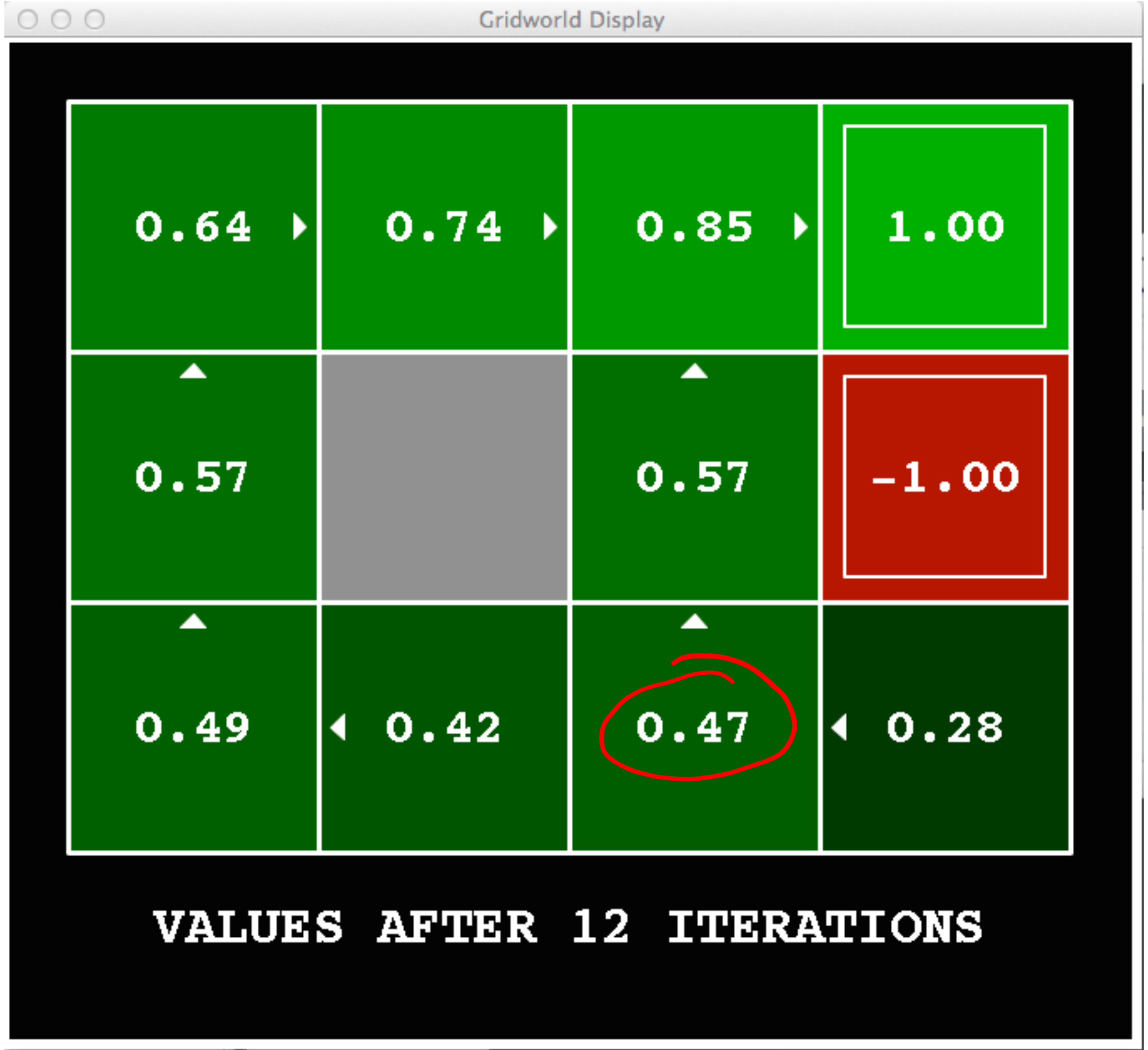
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



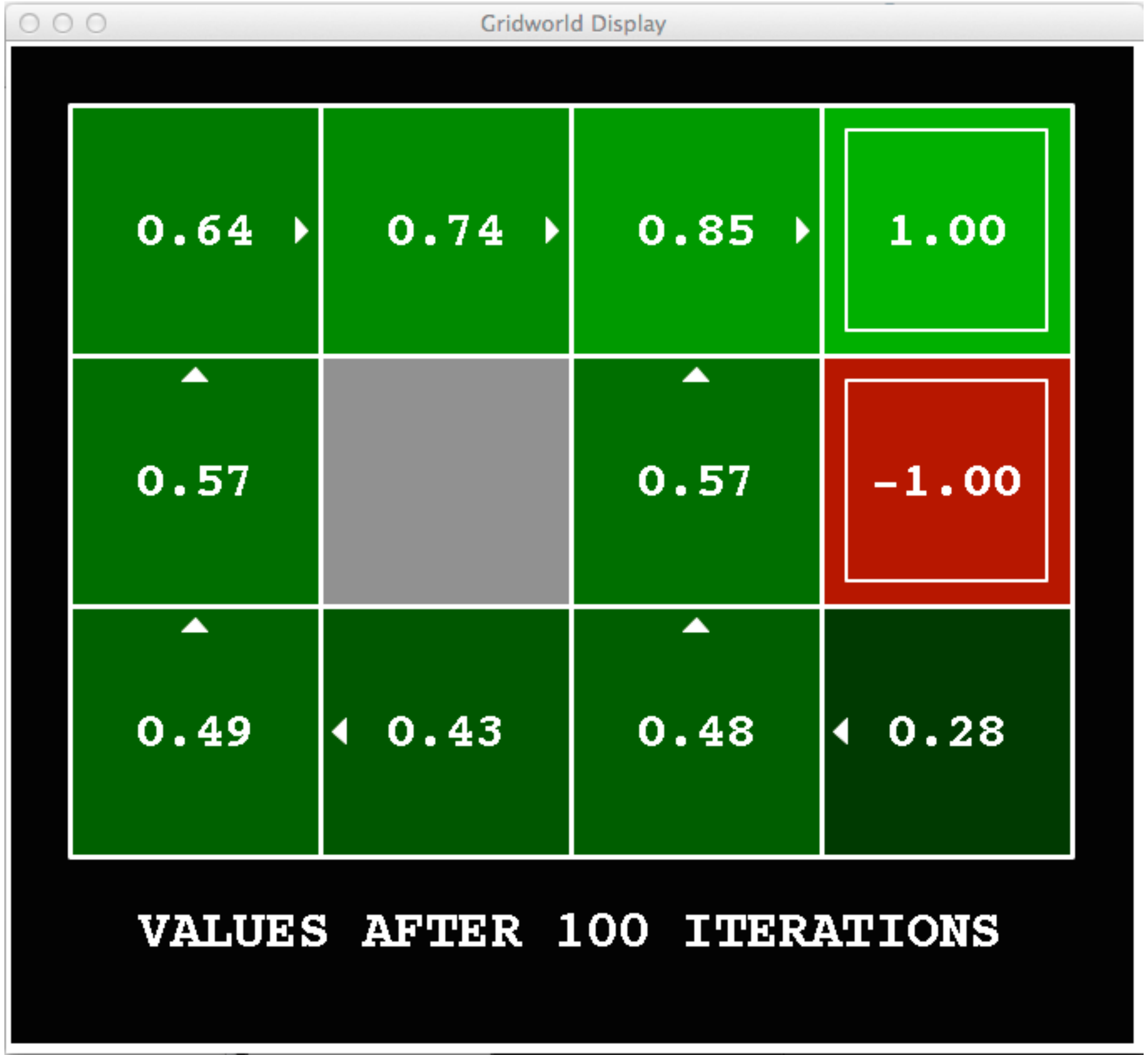
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



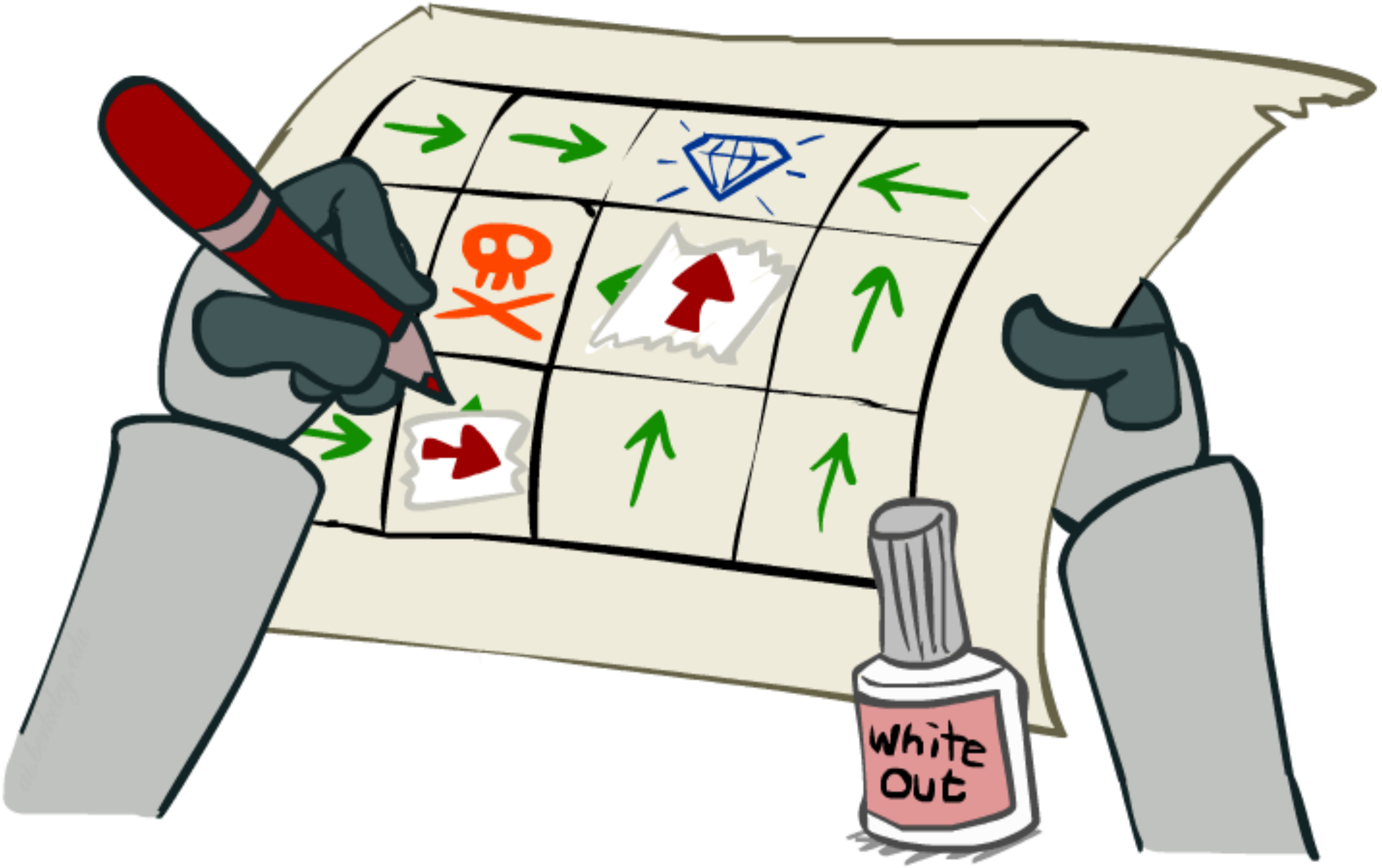
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration



Two Methods for Solving MDPs

Value iteration + policy extraction

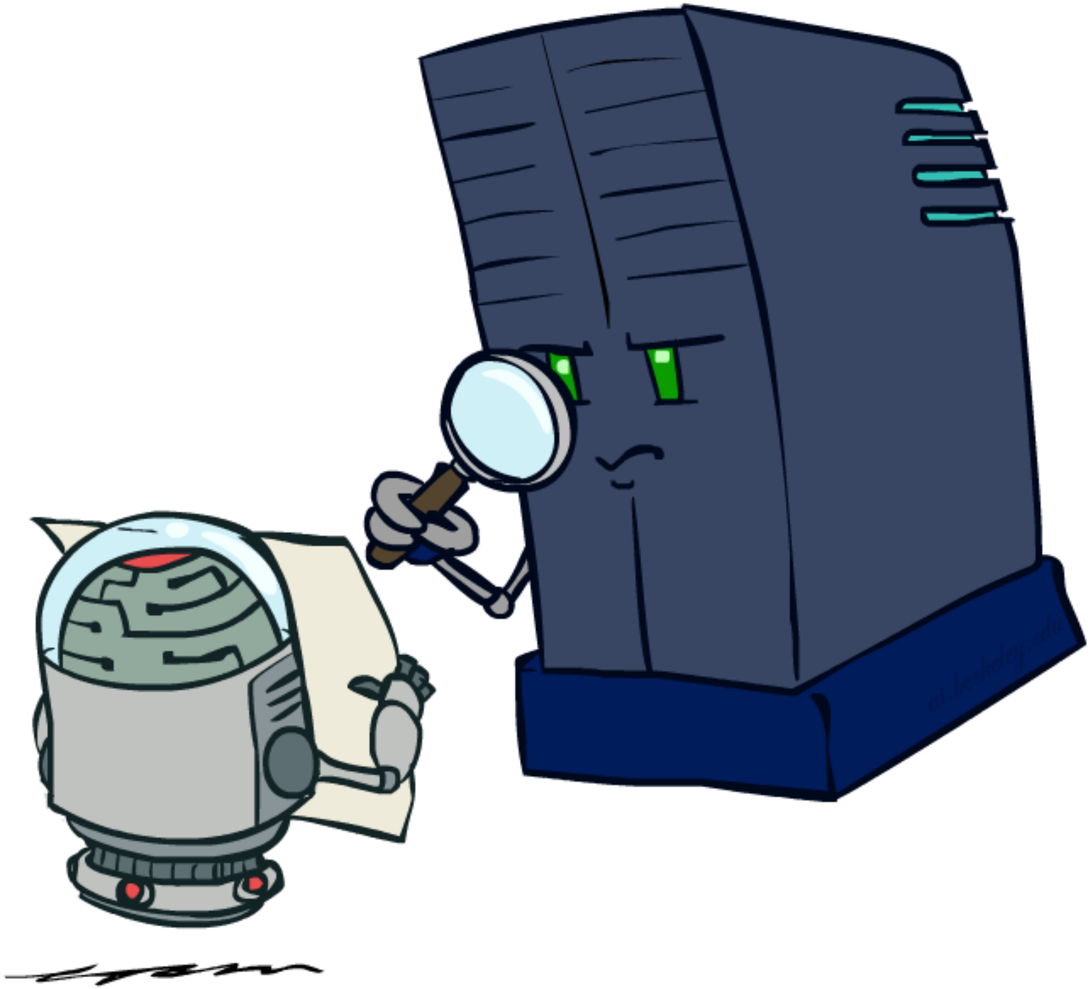
$V_k \rightarrow V_{k+1}$

- 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 $V^* \rightarrow \pi^*$

Policy iteration

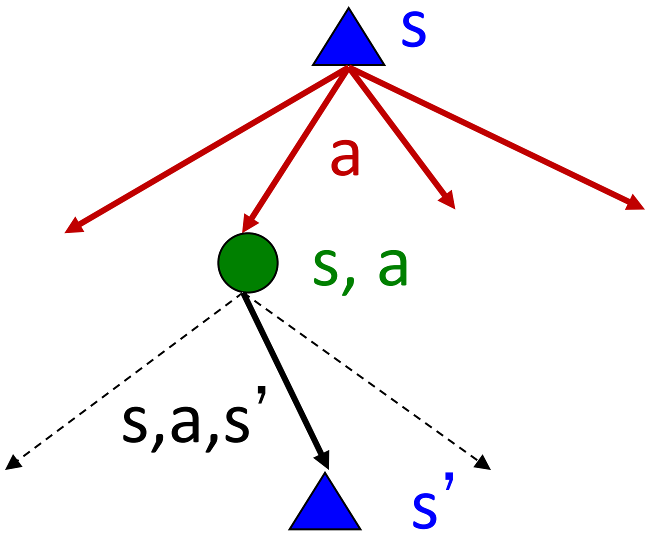
- Step 1: Policy evaluation: calculate values for some fixed policy (not optimal values!) **until convergence** $\pi_0 \rightarrow V^{\pi_0}$
- Step 2: Policy improvement: update policy by running one ply of the Bellman equations using values from policy evaluation $V^{\pi_0} \rightarrow \pi_1$
- Repeat steps until policy converges

Policy Evaluation

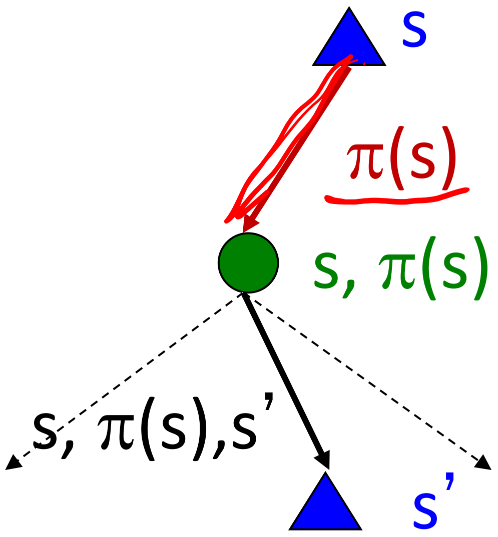


Fixed Policies

Do the optimal action



Do what π says to do



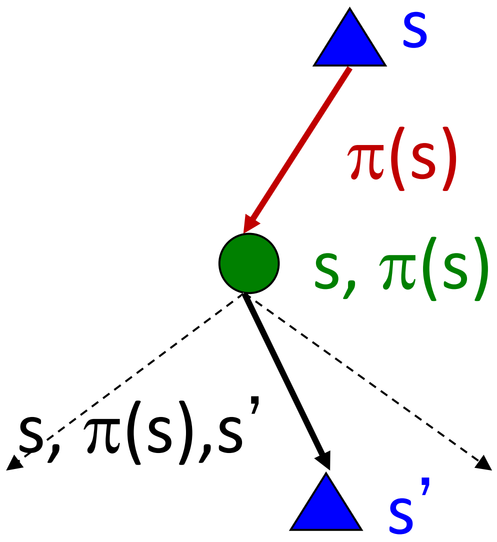
Expectimax trees max over all actions to compute the optimal values

If we fixed some policy 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

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:

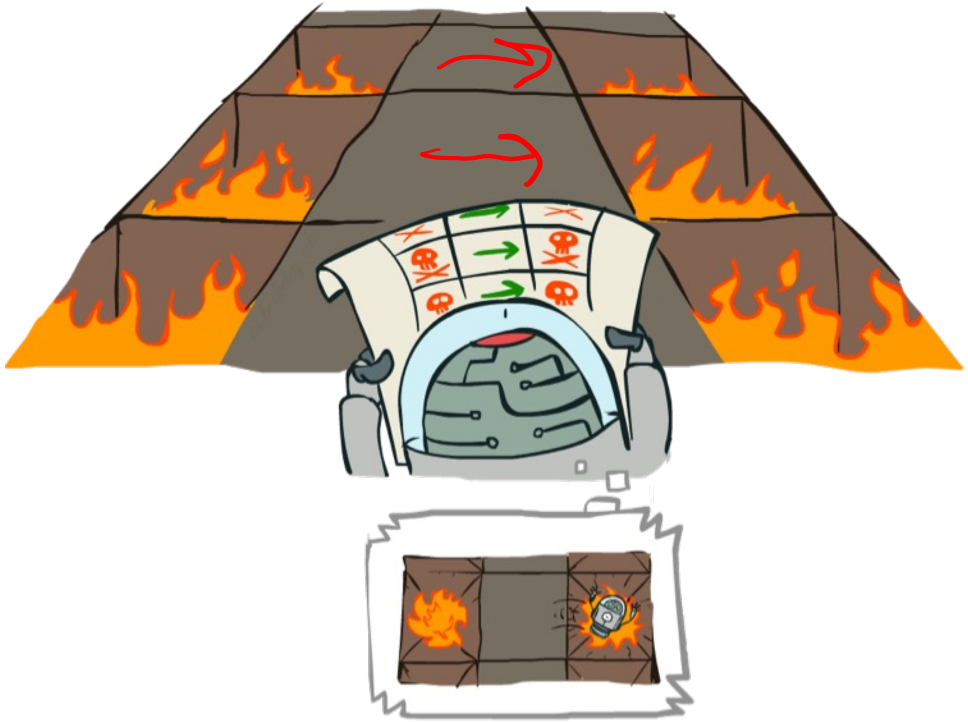
$V^{\pi}(s)$ = expected total discounted rewards starting in s and following π

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

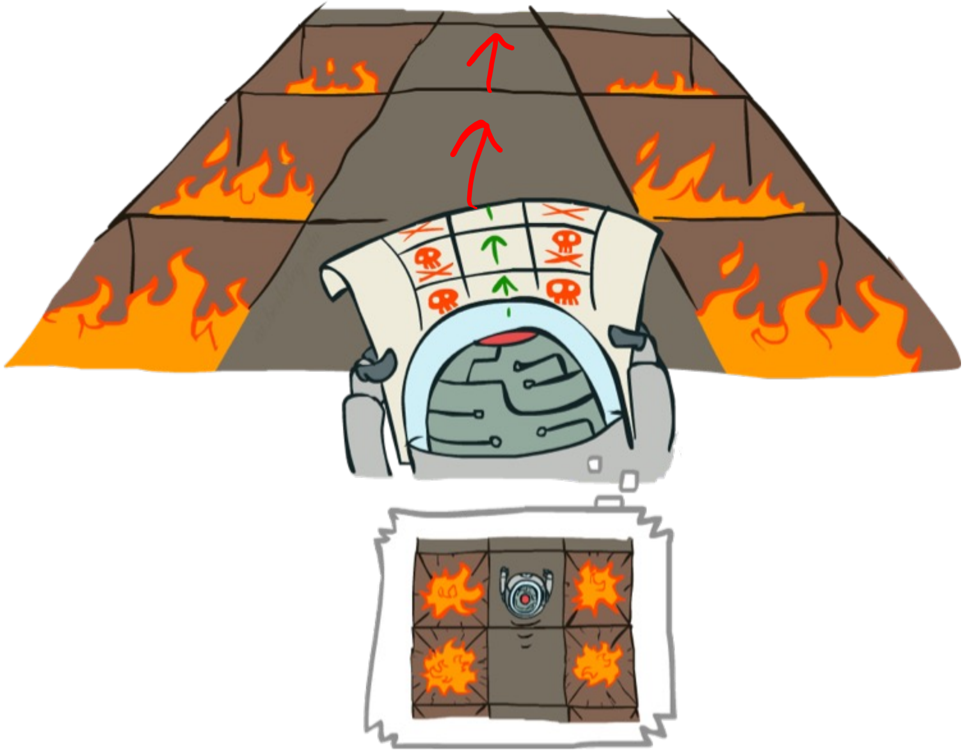
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Example: Policy Evaluation

Always Go Right



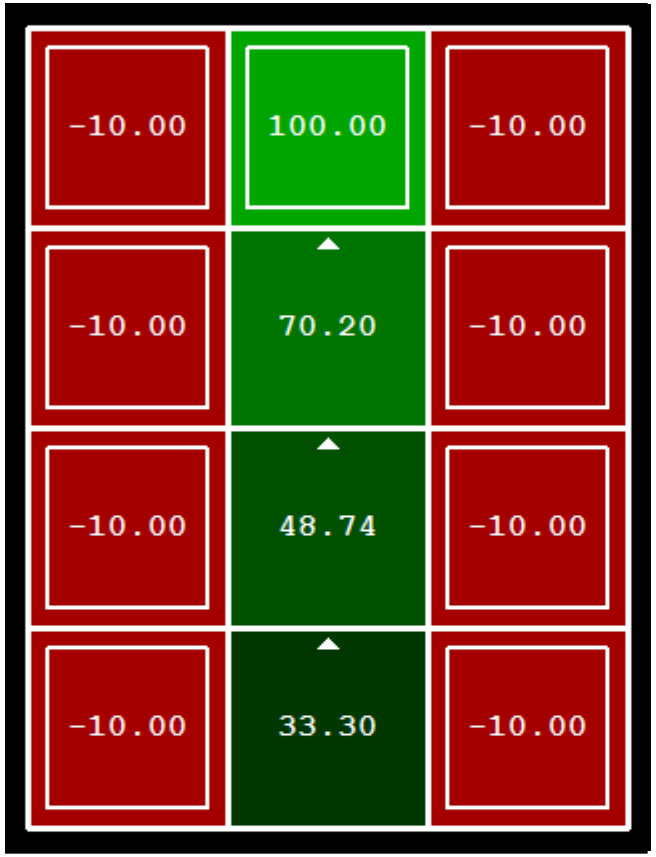
Always Go Forward



Example: Policy Evaluation

Always Go Right

Always Go Forward

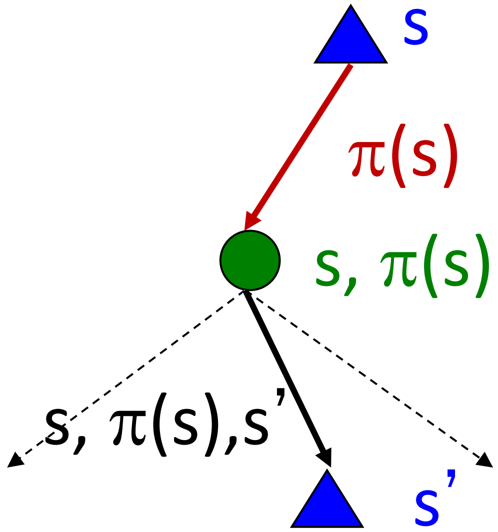


Policy Evaluation

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

Idea 1: Turn recursive Bellman equations into updates

(like value iteration)



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

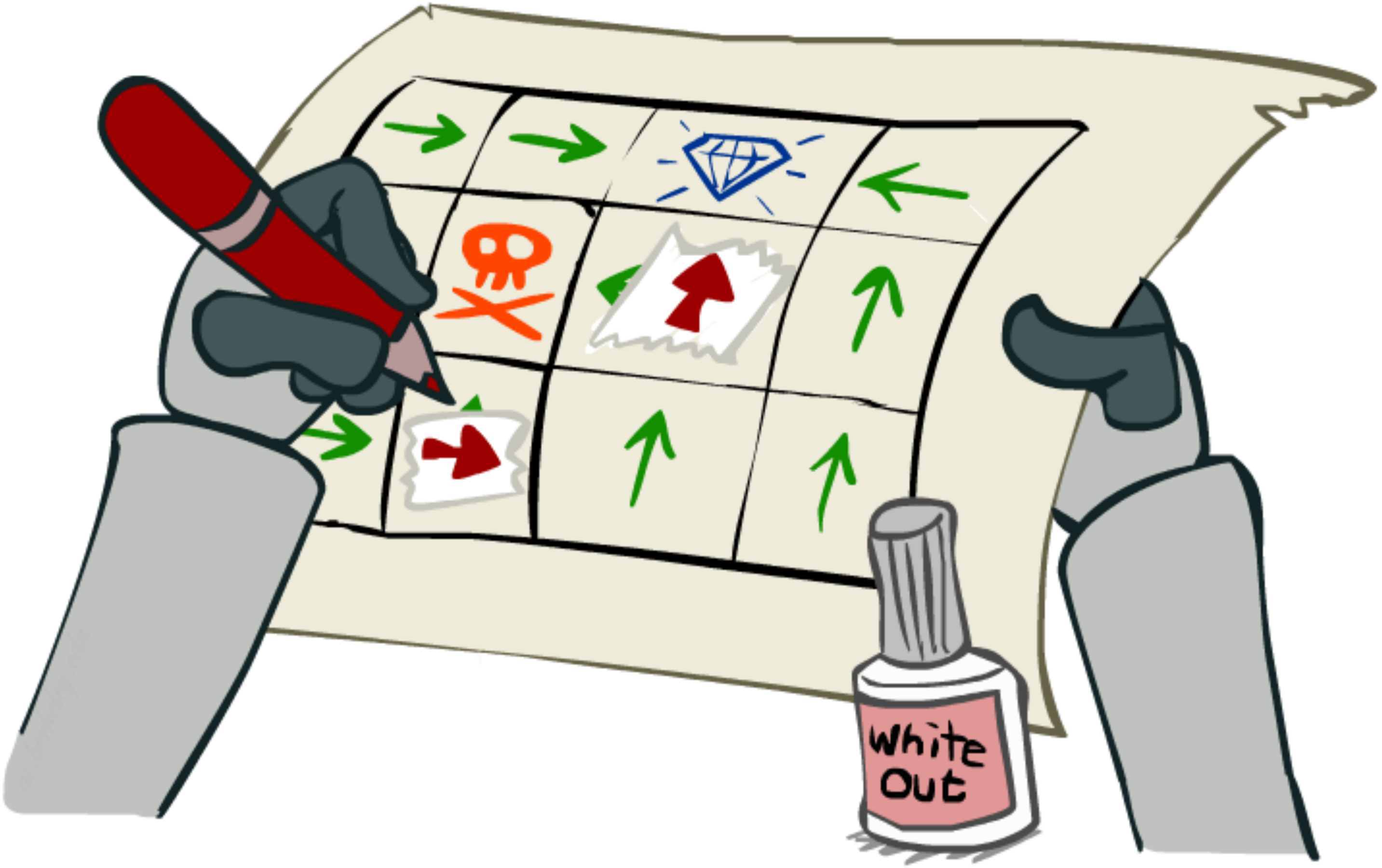
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

Efficiency: O(S²) per iteration

Idea 2: Without the maxes, the Bellman equations are just a linear system

- Solve with your favorite linear system solver

Policy Iteration



Policy Iteration

Alternative approach for optimal values:

- 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

$$\pi_0 \xrightarrow{\text{eval}} V^{\pi_0} \xrightarrow{\text{impro}} \pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2$$

This is policy iteration

- It's still optimal!
- Can converge 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:

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

Two Methods for Solving MDPs

Value iteration + policy extraction

- Step 1: Value iteration:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \text{ , } \forall s$$ **until convergence**

- Step 2: Policy extraction:

$$\pi_V(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')] \text{ , } \forall s$$

Policy iteration

- Step 1: Policy evaluation:

$$V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')] \text{ , } \forall s$$ **until convergence**

- Step 2: Policy improvement:

$$\pi_{new}(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')] \text{ , } \forall s$$

- Repeat steps until policy converges

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 values with fixed policy (each pass is fast because we consider only one action, not all of them; however we do many passes)
- 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

MDP Notation

Standard expectimax: $$V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$$

Bellman equations: $$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

Value iteration: $$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \qquad \forall\, s$$

Q-iteration: $$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$$

Policy extraction: $$\pi_V(s) = \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \qquad \forall\, s$$

Policy evaluation: $$V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')], \qquad \forall\, s$$

Policy improvement: $$\pi_{new}(s) = \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \qquad \forall\, s$$

MDP Notation

Standard expectimax: $V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$

Bellman equations: $V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$

Value iteration: $V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \quad \forall\, s$

Q-iteration: $Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$

Policy extraction: $\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \quad \forall\, s$

Policy evaluation: $V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')], \quad \forall\, s$

Policy improvement: $\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \quad \forall\, s$

MDP Notation

Standard expectimax: $V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$

Bellman equations: $V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$

Value iteration: $V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \quad \forall\, s$

Q-iteration: $Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$

Policy extraction: $\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \quad \forall\, s$

Policy evaluation: $V^{\pi}_{k+1}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V^{\pi}_k(s')], \quad \forall\, s$

Policy improvement: $\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \quad \forall\, s$

MDP Notation

Standard expectimax: $V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$

Bellman equations: $V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$

Value iteration: $V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \quad \forall\, s$

Q-iteration: $Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$

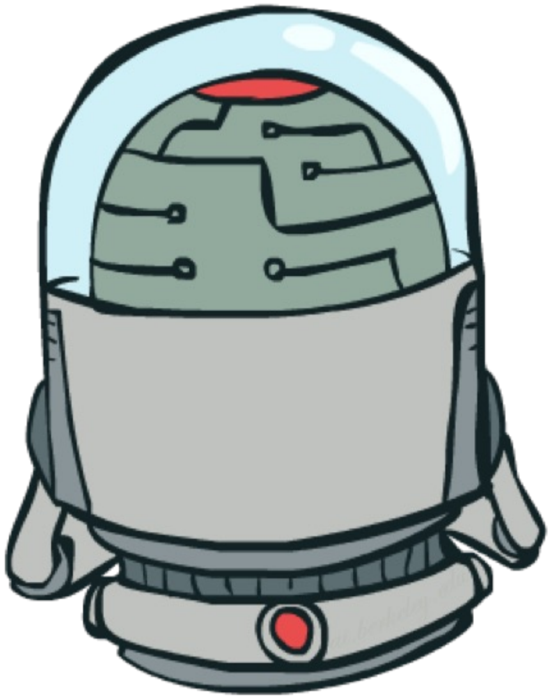
Policy extraction: $\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \quad \forall\, s$

Policy evaluation: $V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')], \quad \forall\, s$

Policy improvement: $\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \quad \forall\, s$

Next Time: Reinforcement Learning!

Double Bandits



Double-Bandit MDP

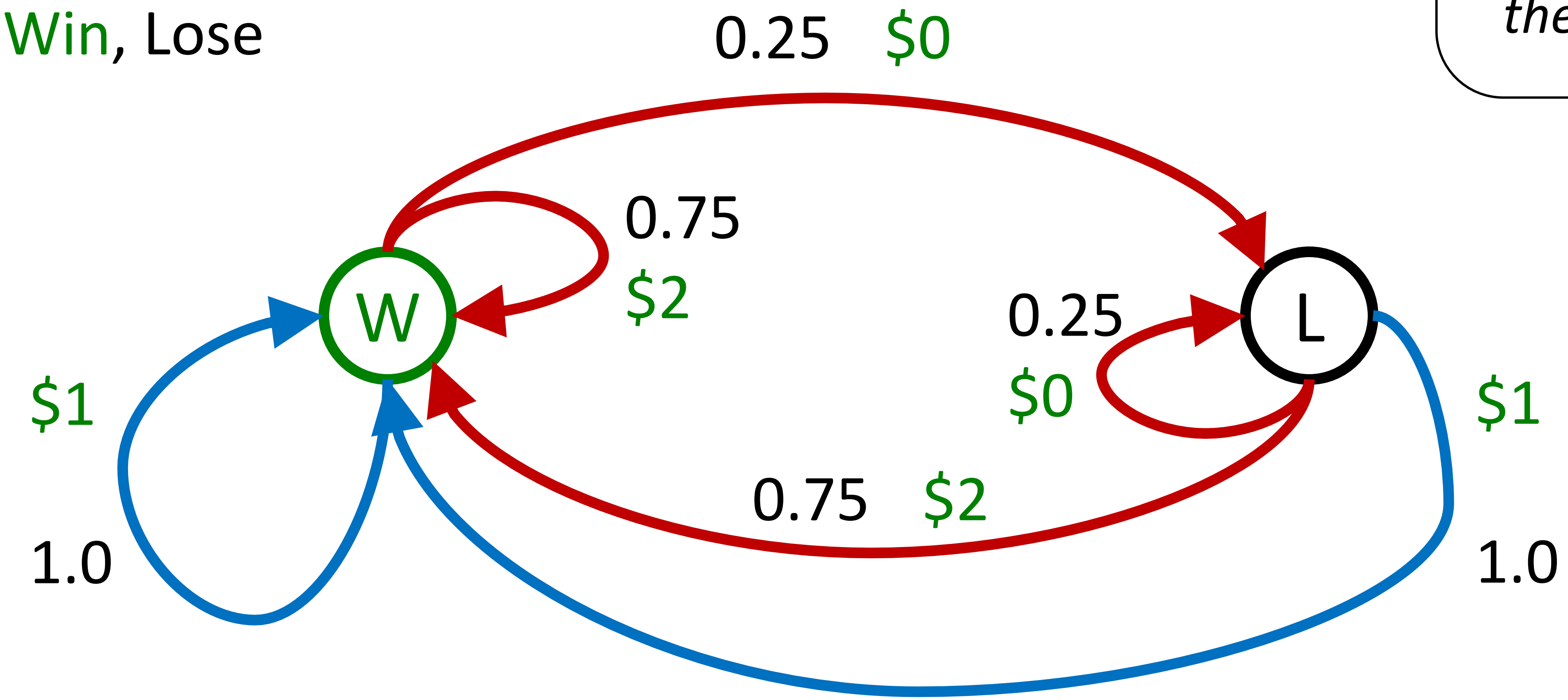
Actions: *Blue*, *Red*

States: Win, Lose

No discount

100 time steps

Both states have the same value



Offline Planning

Solving MDPs is offline planning

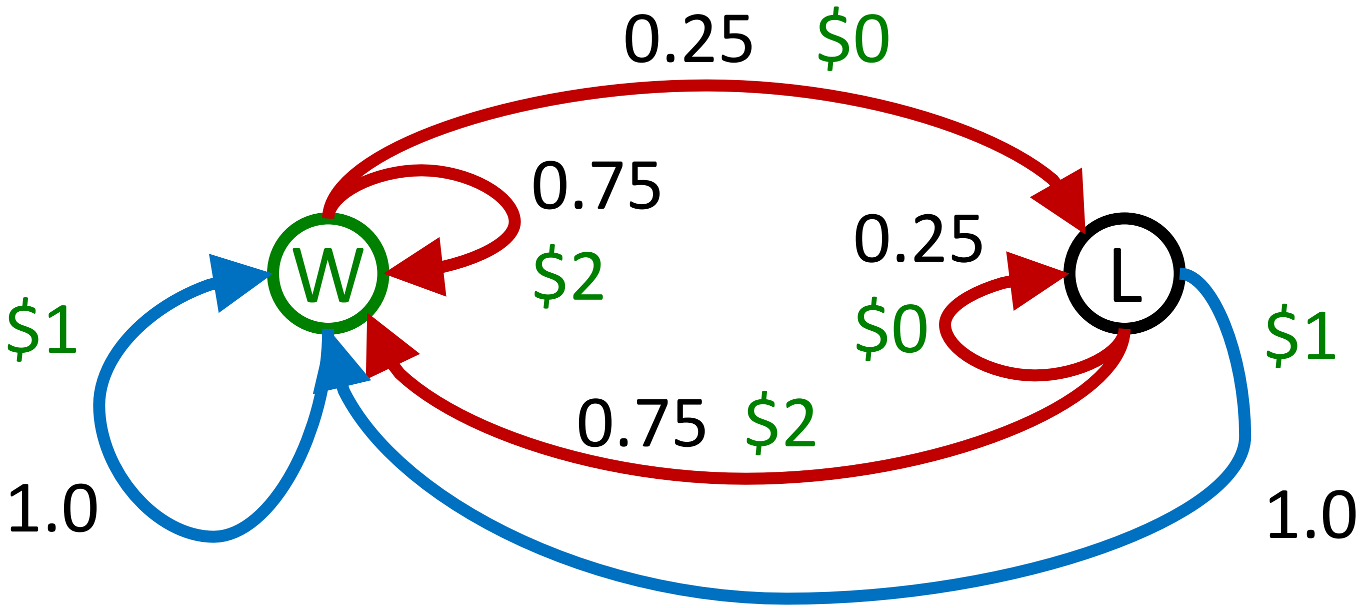
- You determine all quantities through computation
- You need to know the details of the MDP
- You do not actually play the game!

No discount

100 time steps

Both states have the same value

	Value
Play Red	150
Play Blue	100



Slide: ai.berkeley.edu

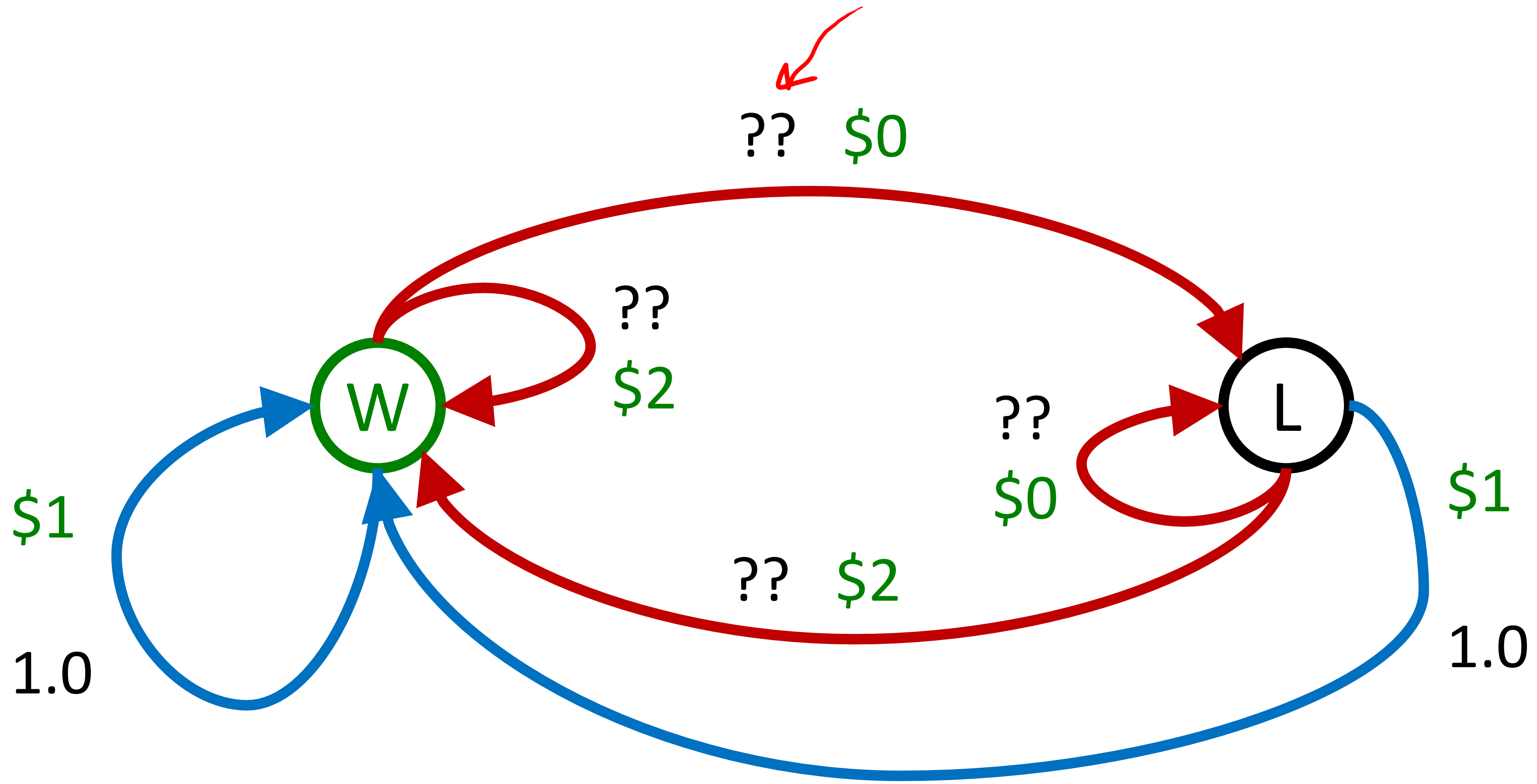
Let's Play!

$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