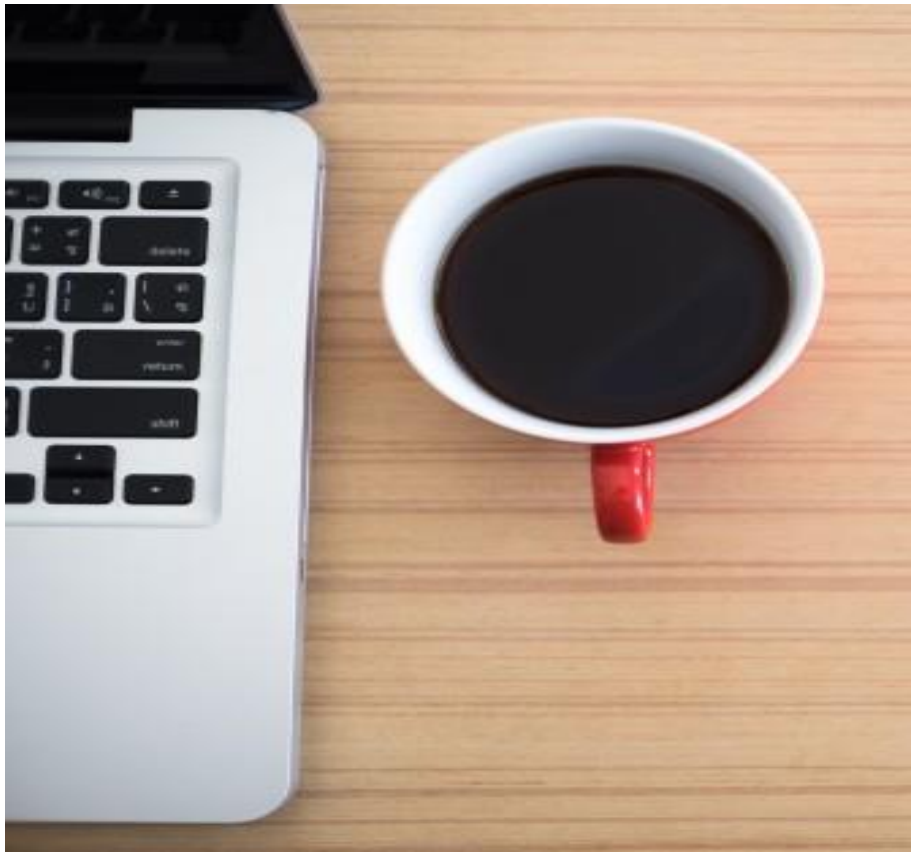
Fourth Industrial Summer School

Advanced Machine Learning

Reinforcement Learning-part1

Session Objectives

- ✓Introduction
- ✓Foundations (MDP)
- ✓Reinforcement Learning



Reinforcement Learning

Introduction

Reinforcement Learning

- What is it?
- Supervised learning vs. reinforcement learning

Example: Balancing act

Example: Toddler Robot

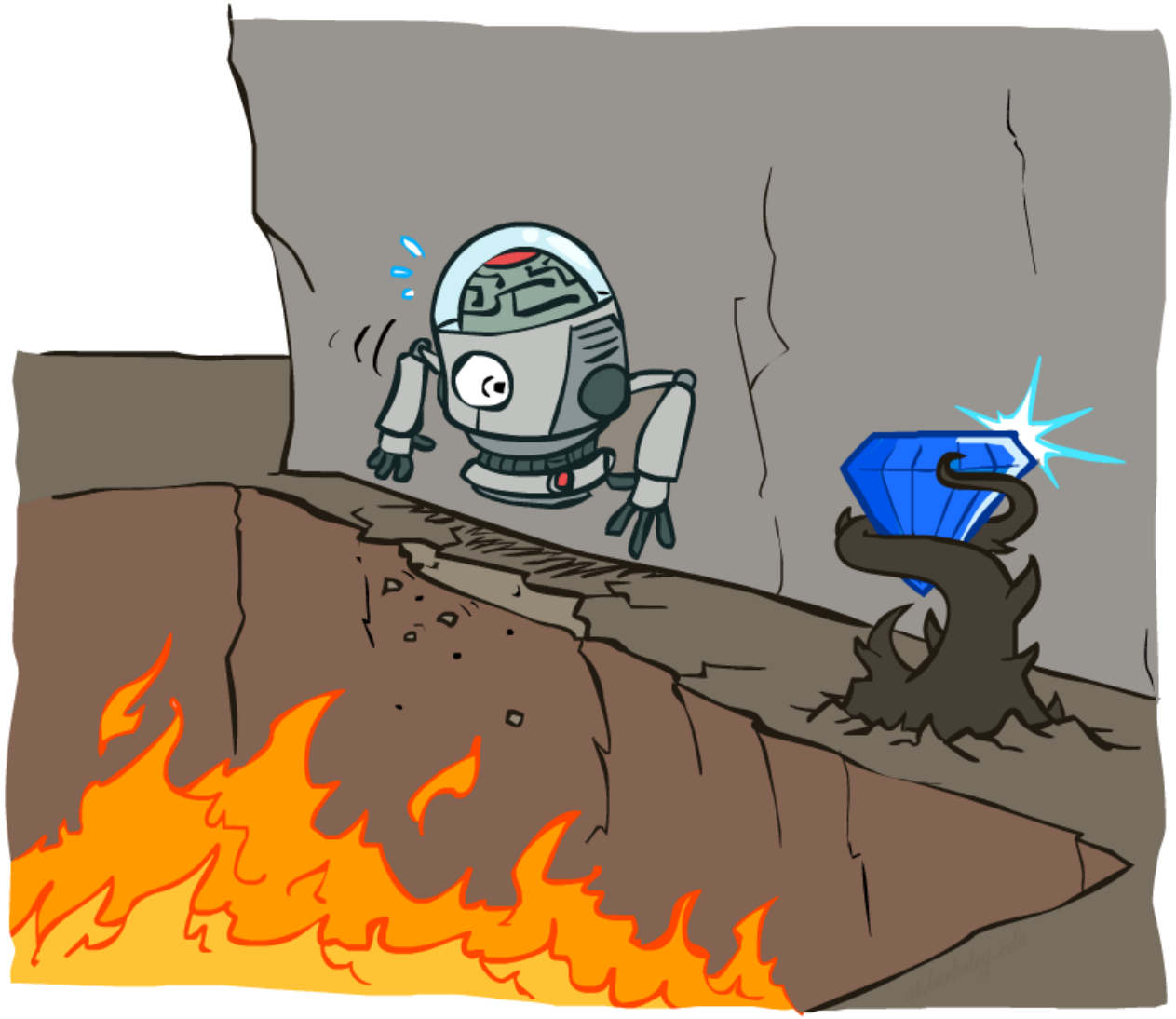
[Tedrake et al., 2005]

Example-Playing a game

https://www.youtube.com/watch?v=V1eYniJ0Rnk

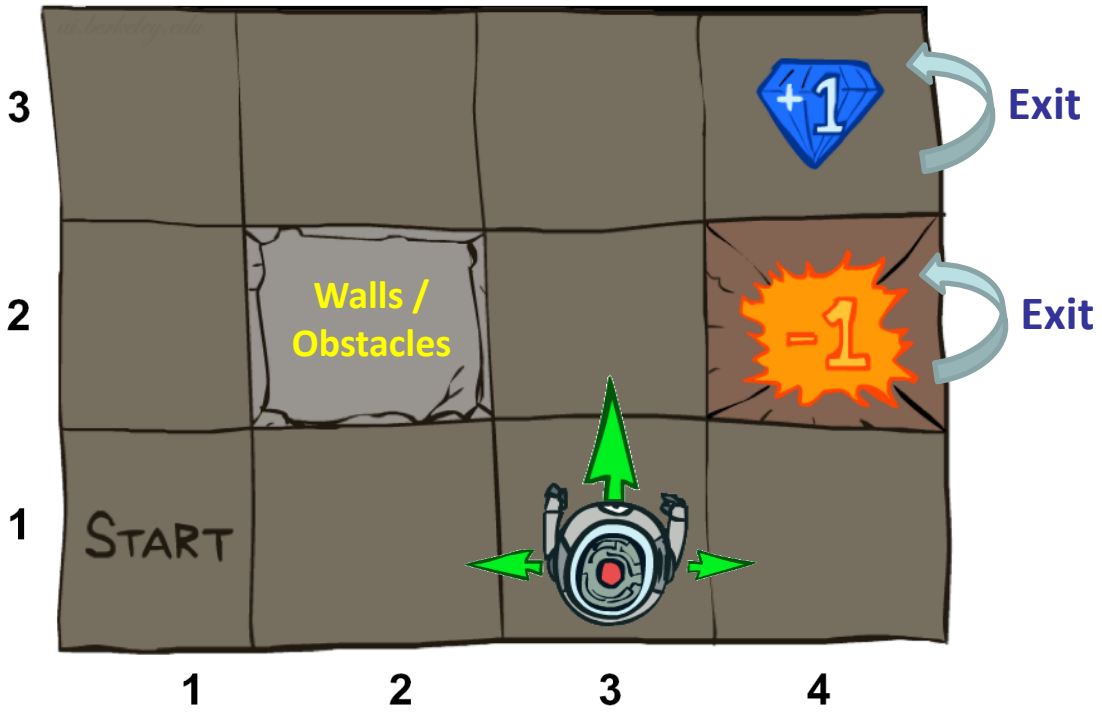
Markov Decision Process (MDPs)

Non-Deterministic Search



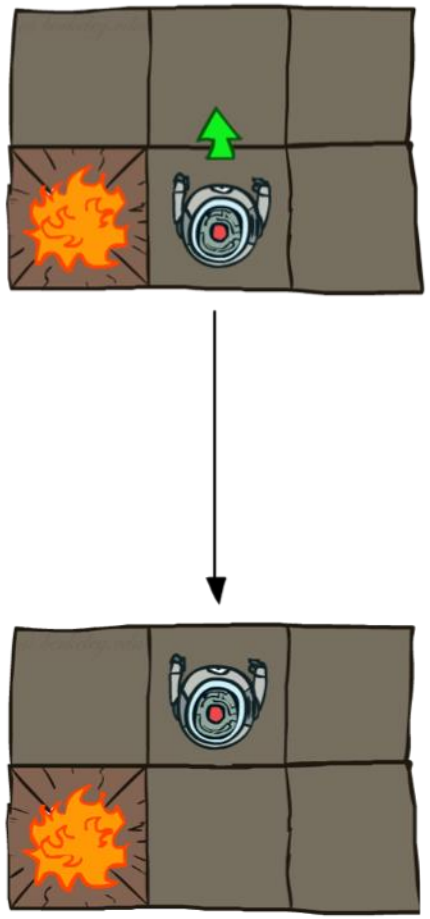
Example: Grid World

- A ***maze-like*** problem
  - The agent lives in a grid
  - Walls block the agent's path
  - At terminal (absorbing) states, the agent takes the exit actions and stays there with no more rewards accumulated!
- **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
  - **Note:** 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**

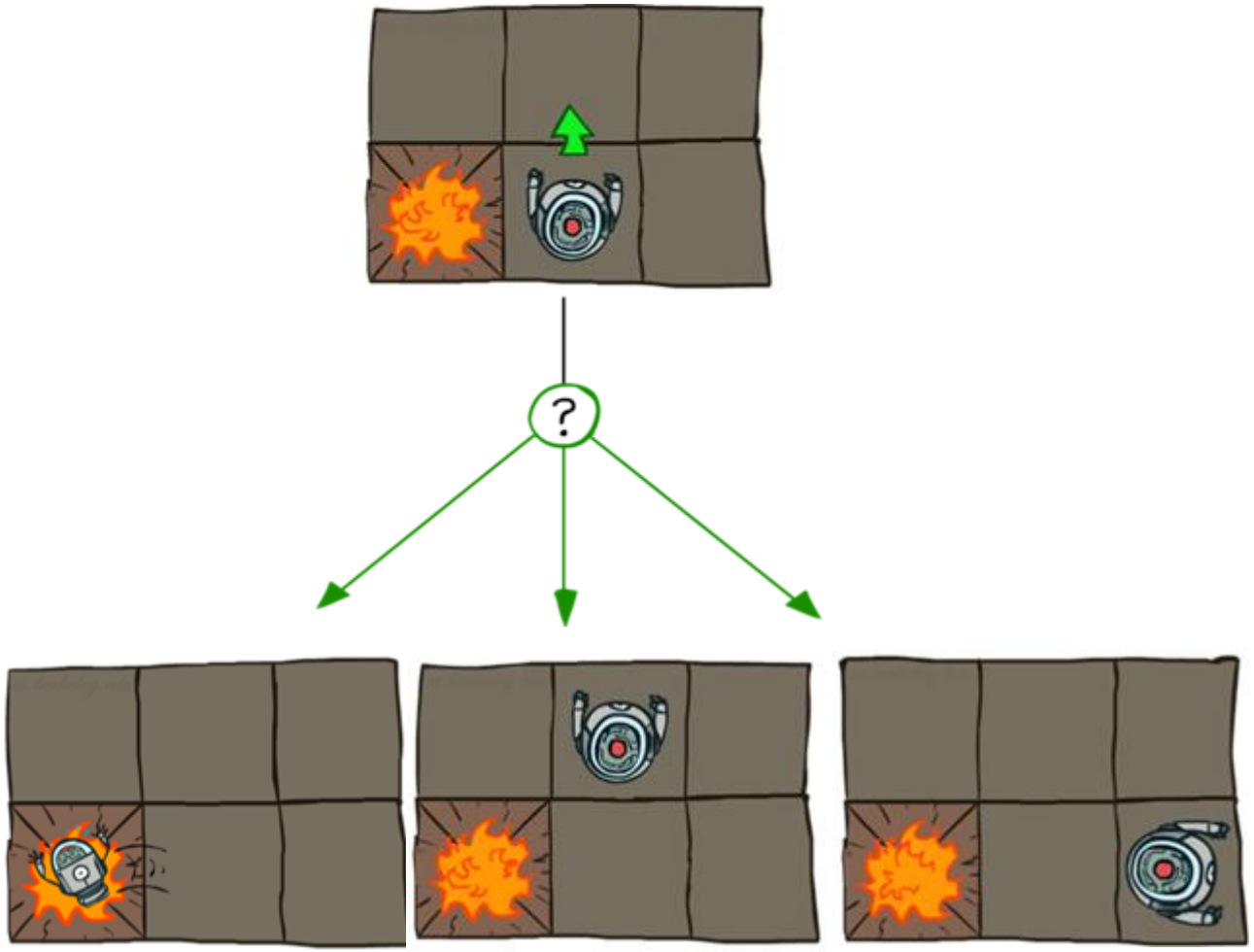


Grid World Actions

Deterministic Grid World

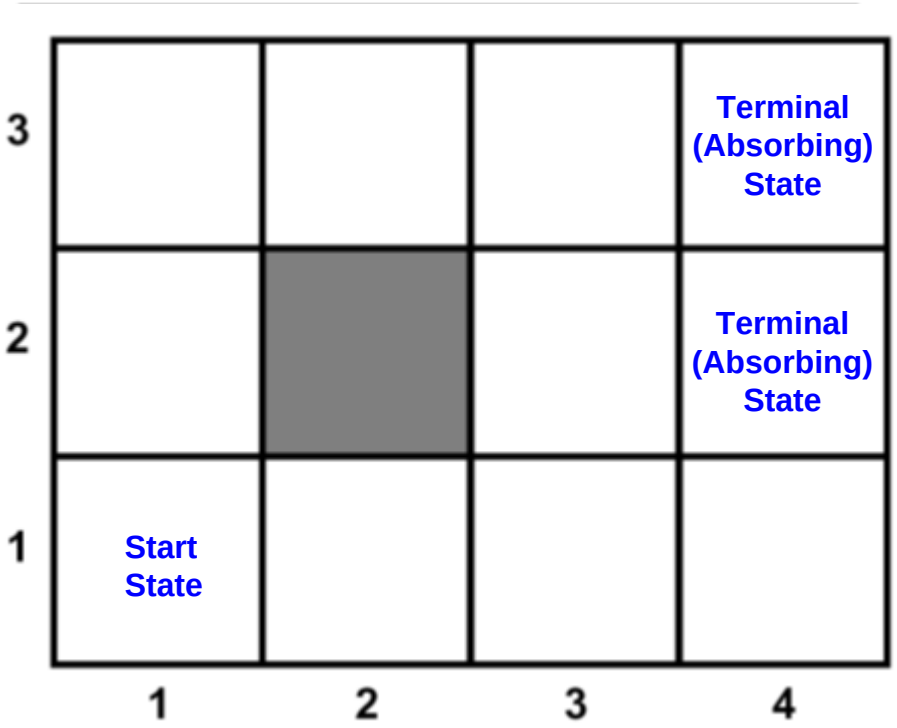


Stochastic Grid World



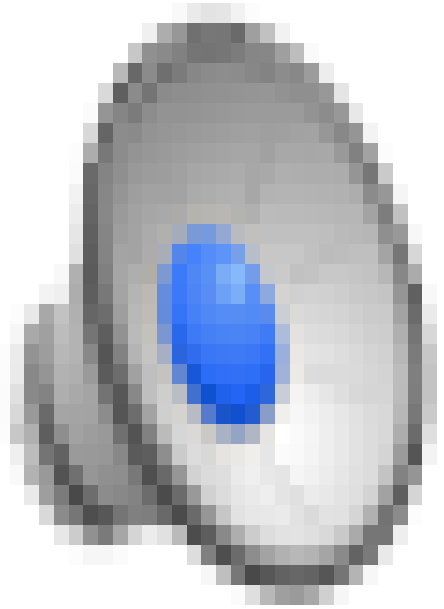
Markov Decision Processes (MDPs)

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



- **MDPs are non-deterministic search problems**
  - One way to solve them is with expectimax search
  - However, there are better ways to solve them!

MDP Agent Using Manual Actions (Cont’d)



What is Markov about MDPs?

- "**Markov**" generally means that **given the present state, the future and the past are independent**

- For MDPs, "**Markov**" means **action outcomes depend only on the current state**:

Future Present Past (Infinite history)

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

Future Present

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

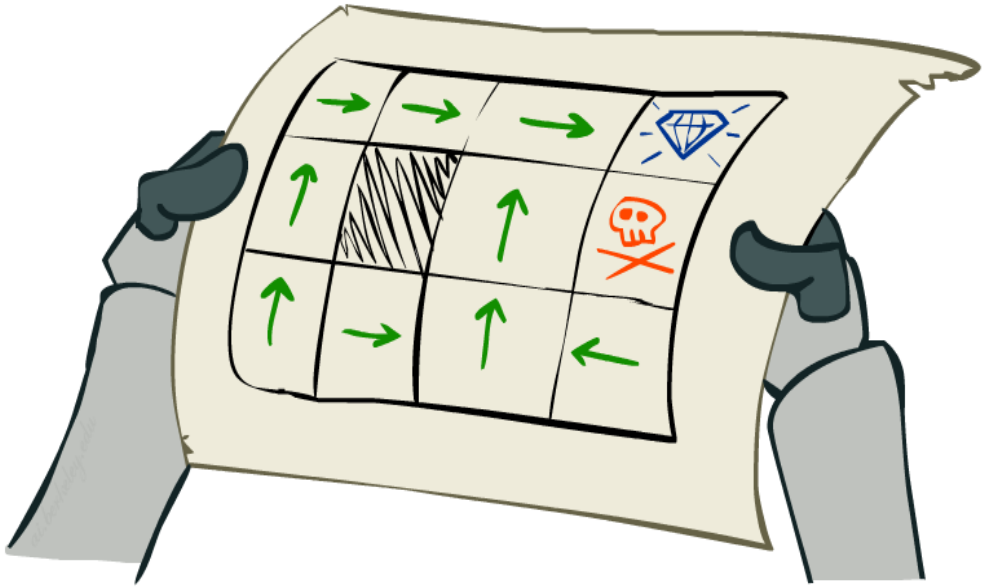
Andrey Markov (1856-1922)

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

Policies

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

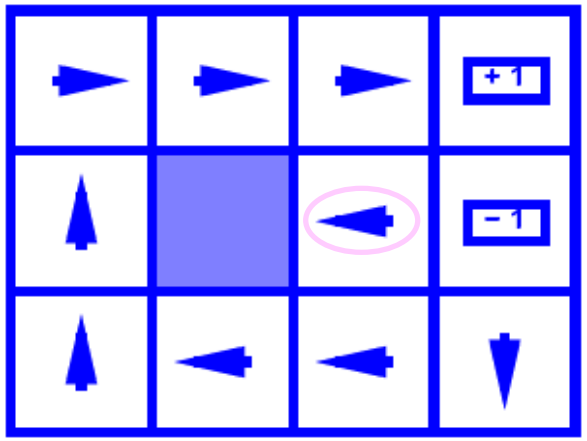
- For MDPs, we want an optimal ***policy $\pi^*: S \rightarrow A$***
  - A policy π gives an action for each state
  - An **optimal policy** is one that maximizes the ***expected utility*** **if followed**
  - An explicit policy defines a **reflex agent**



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

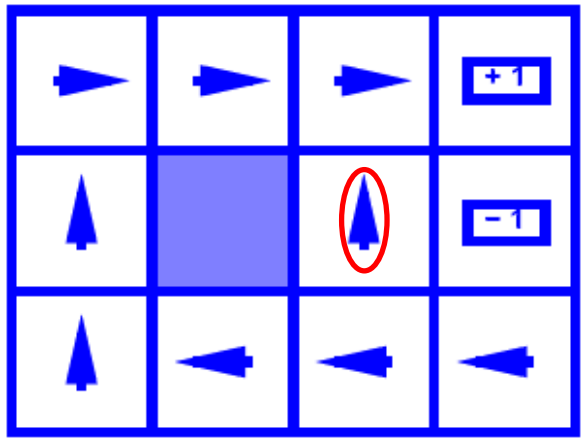
Optimal Policies

Small penalties so take no risk in bumping onto the worst state!



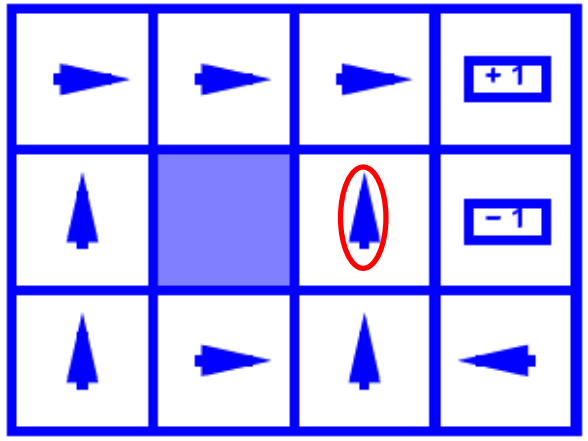
R(s) = -0.01

Higher penalties so take some risk to reduce losses!



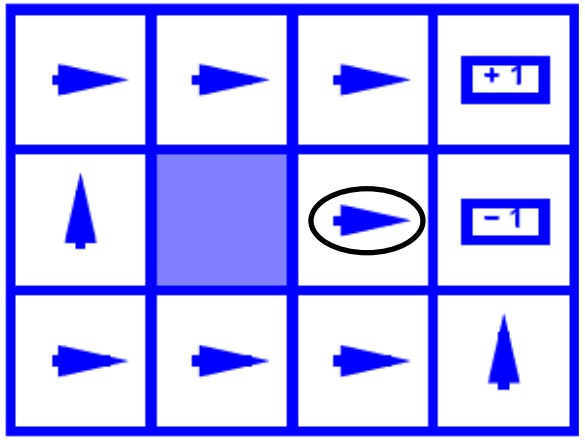
R(s) = -0.03

Higher penalties so take some risk to reduce losses!



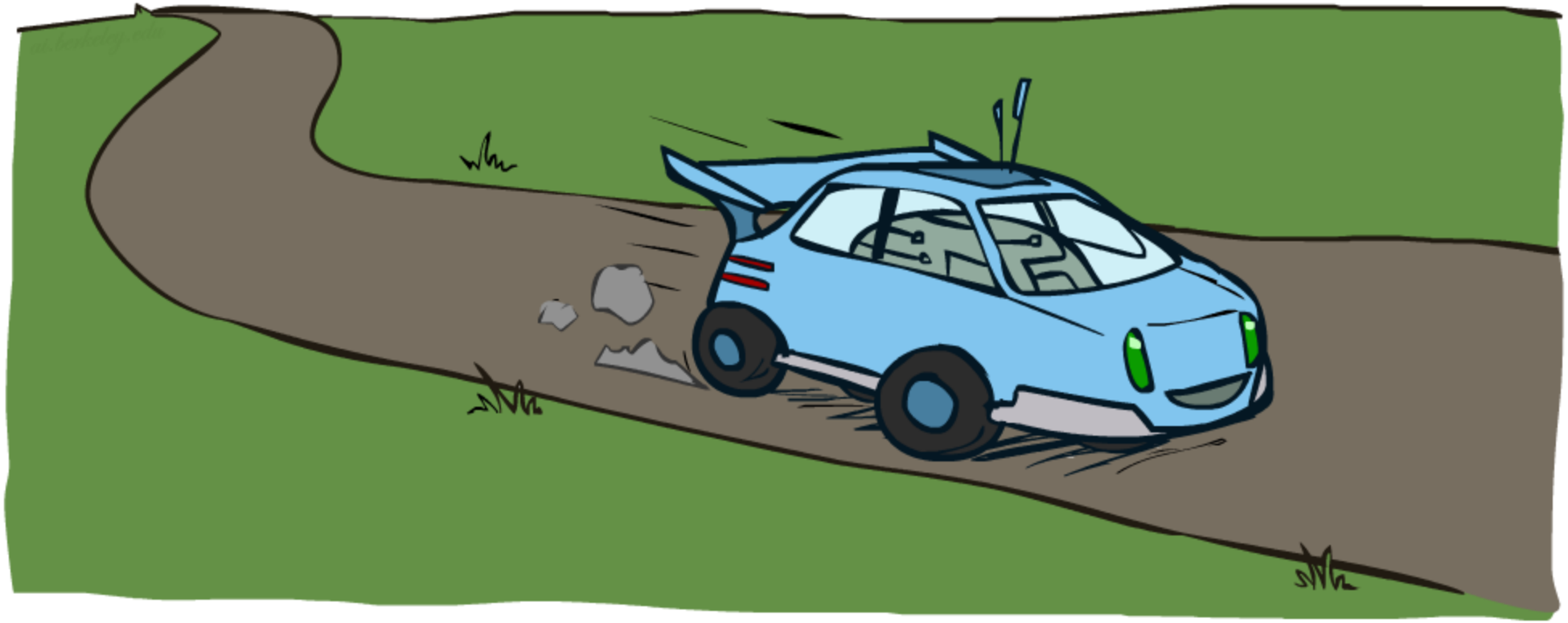
R(s) = -0.4

Live hell! Better end your life!



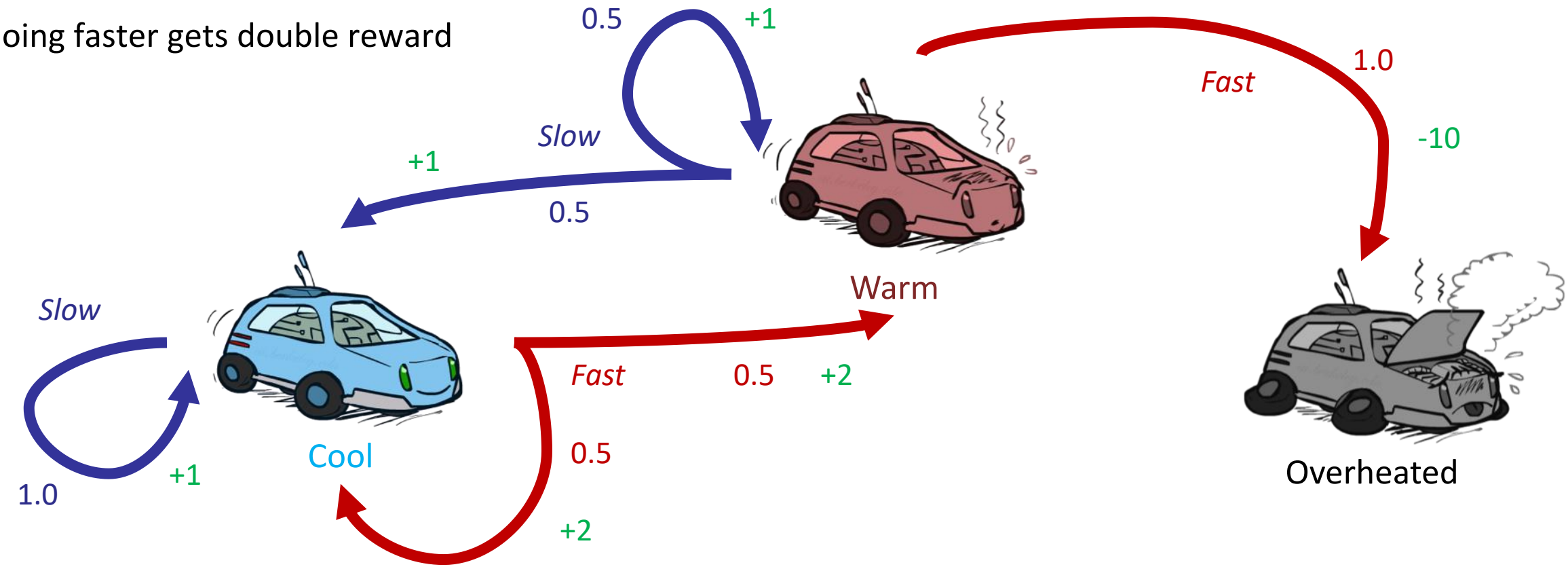
R(s) = -2.0

Example: Racing

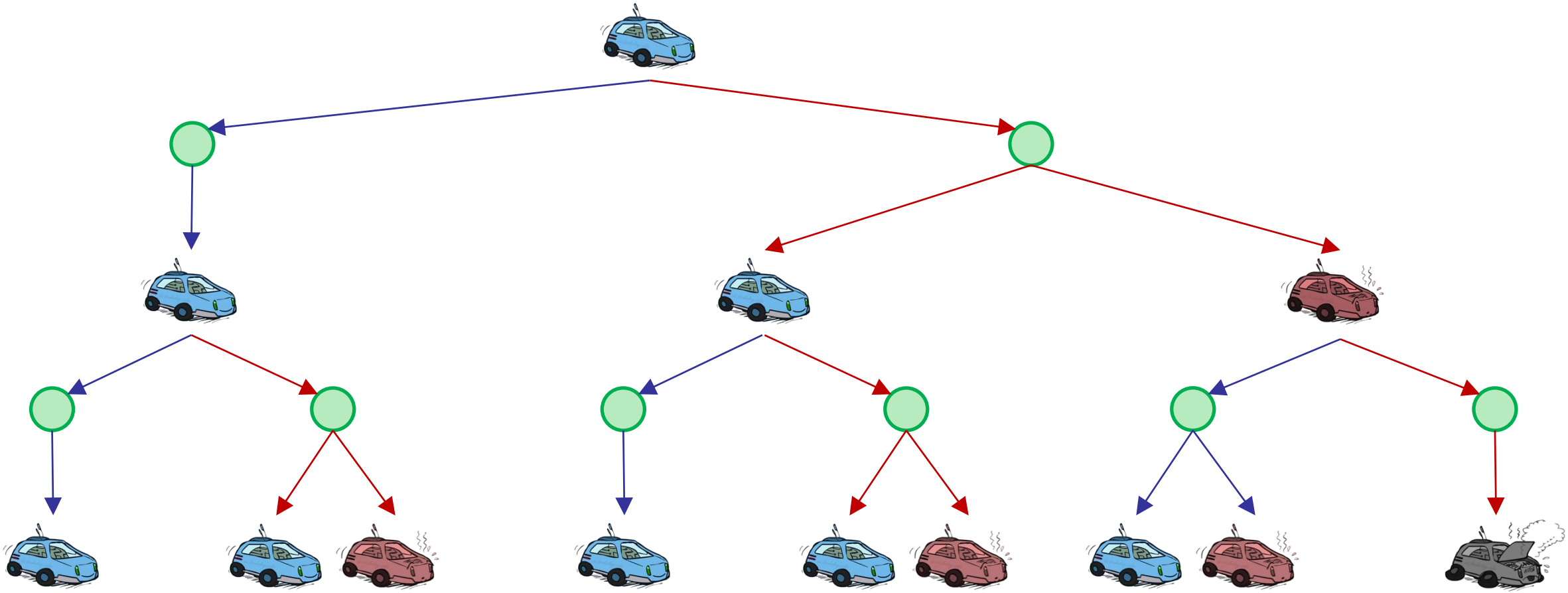


Example: Racing

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



Racing Search Tree

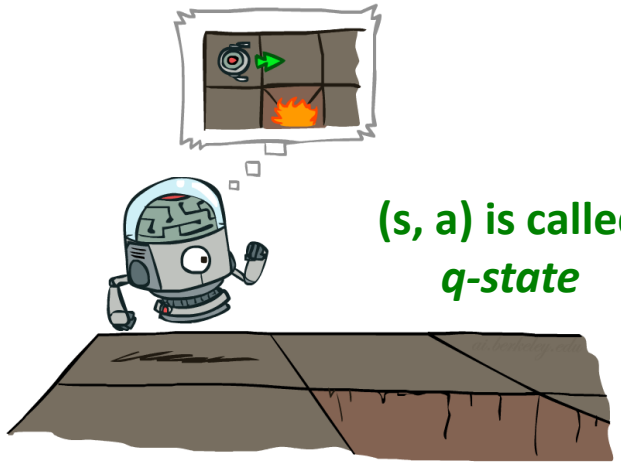


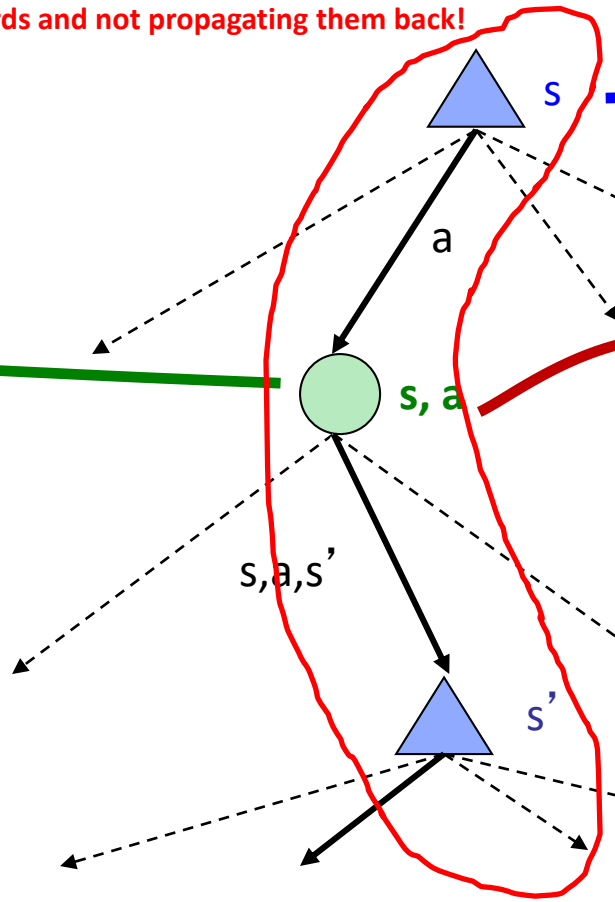
Depth-2 Expectimax

MDP Search Trees

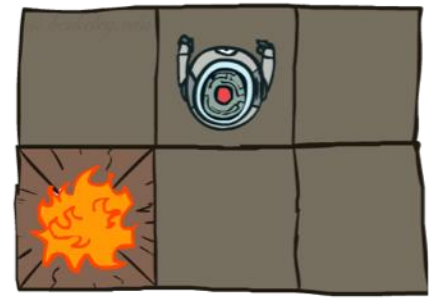
- Each MDP state projects an expectimax-like search tree

Summing up rewards and not propagating them back!





s is a *state*



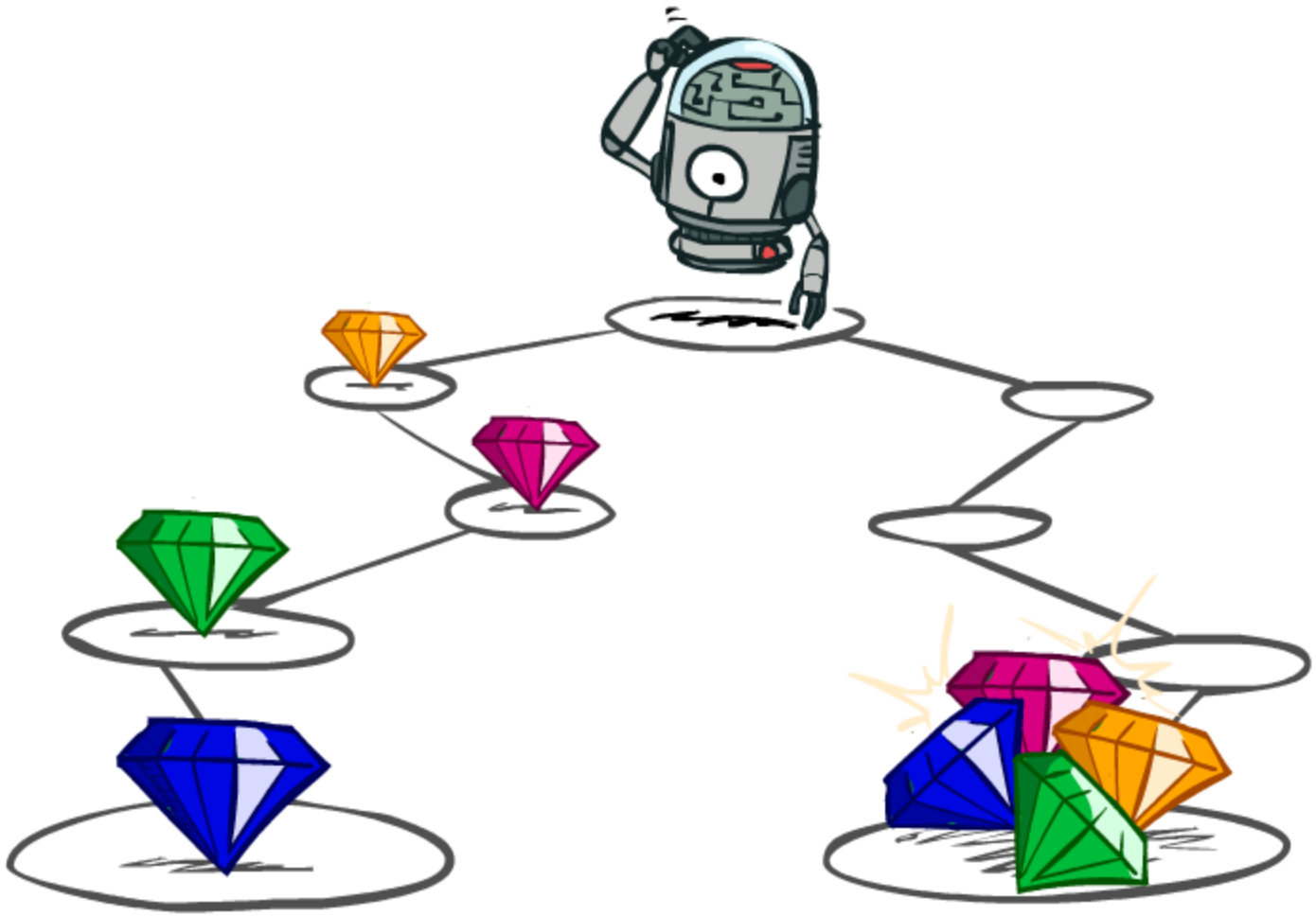
(s, a) is called *q-state*

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

T(s,a,s') = P(s'|s,a) : Conditional probability

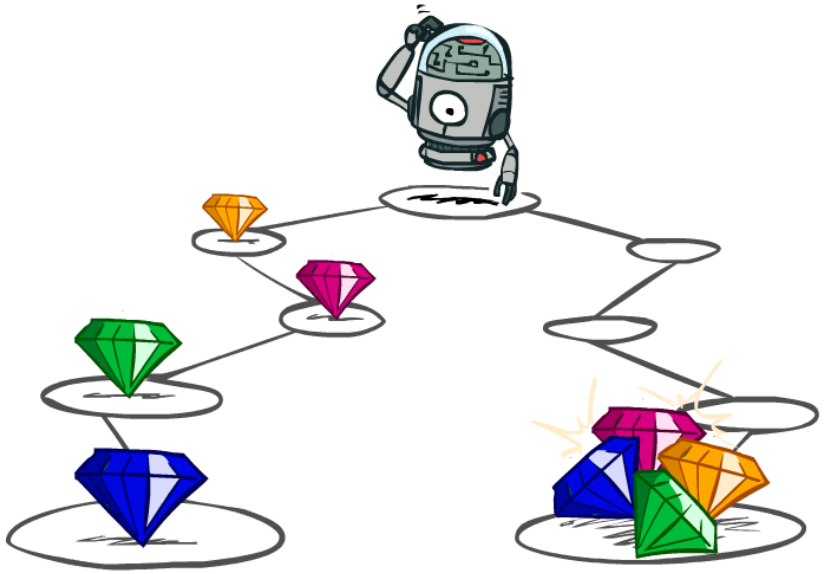
R(s,a,s')

Utilities of Sequences



Utilities of Sequences

- What preferences should an agent have over reward sequences?
- More or less? [1, 2, 2] or [2, 3, 4]
- Now or later? [0, 0, 1] or [1, 0, 0]



Discounting

- It is reasonable to maximize the sum of rewards
- It is also reasonable to prefer rewards now to rewards later
- One solution: Values of rewards decay exponentially

1

Worth Now

γ

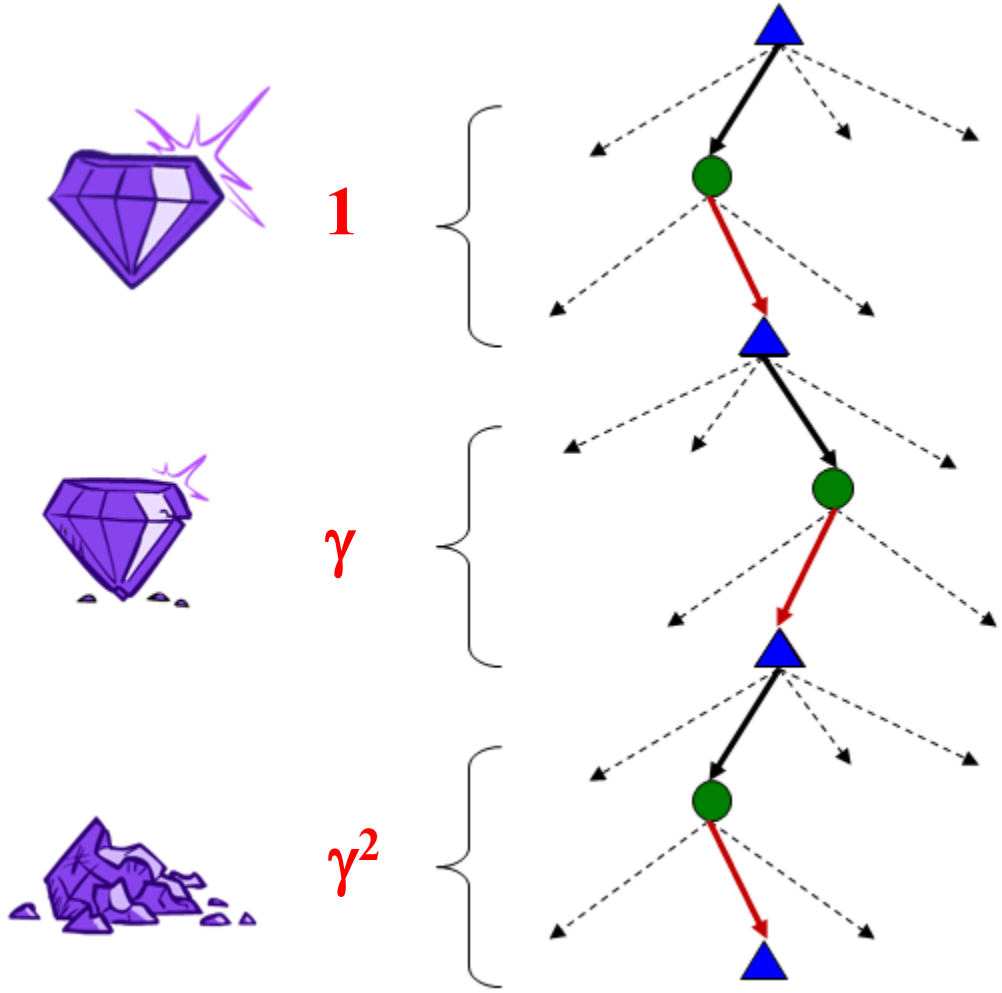
Worth Next Step

γ^2

Worth In Two Steps

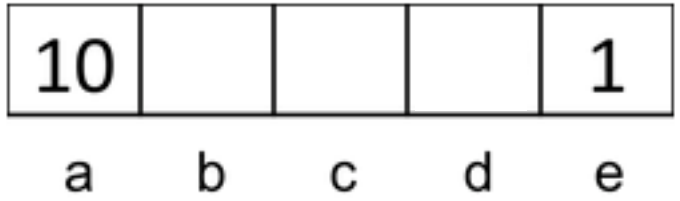
Discounting (Cont'd)

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also **helps our algorithms converge**
- **Example:** Discount of $\gamma = 0.5$
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



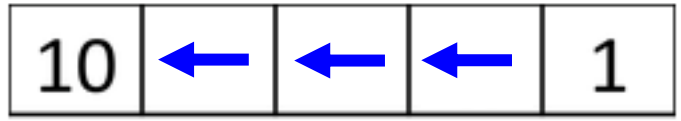
Quiz: Discounting

- Given:

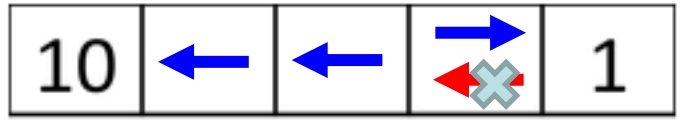


  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: Deterministic

- Quiz 1: For $\gamma = 1$, what is the optimal policy?



- Quiz 2: For $\gamma = 0.1$, what is the optimal policy?

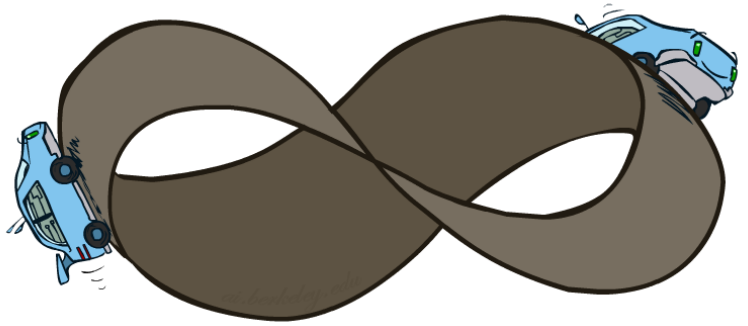


- Quiz 3: For which γ are West and East equally good when in state ***d***?

Infinite Utilities?!

- **Problem:** What if the game lasts forever? Do we get infinite rewards?

- **Solutions:**
  - **Finite horizon:** (Similar to depth-limited search)
    - Terminate episodes after a fixed ***T*** steps (e.g., game lifetime)
    - Gives **nonstationary policies** (π depends on time left)
  - **Discounting:** ***0 < γ < 1***
    - Smaller γ means smaller "***horizon***" – Shorter term focus

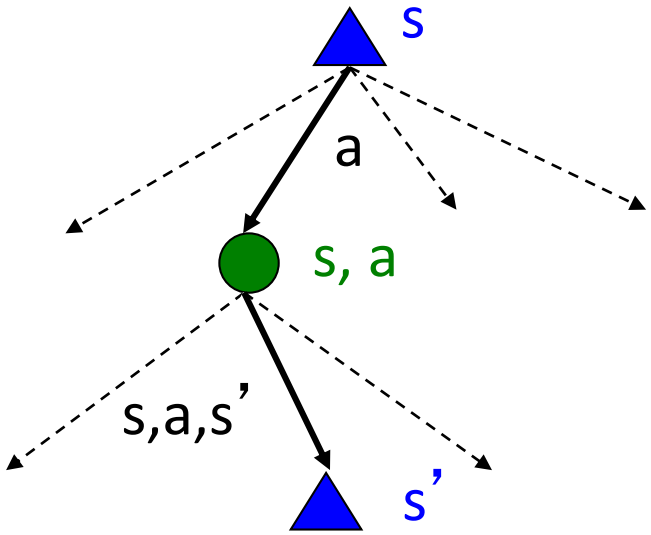


$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

$$\sum_{t=0}^{\infty} \gamma^t \cdot r_t \leq \sum_{t=0}^{\infty} \gamma^t \cdot R_{\max} \leq R_{\max} \cdot \sum_{t=0}^{\infty} \gamma^t \leq R_{\max} \frac{1}{1-\gamma}$$

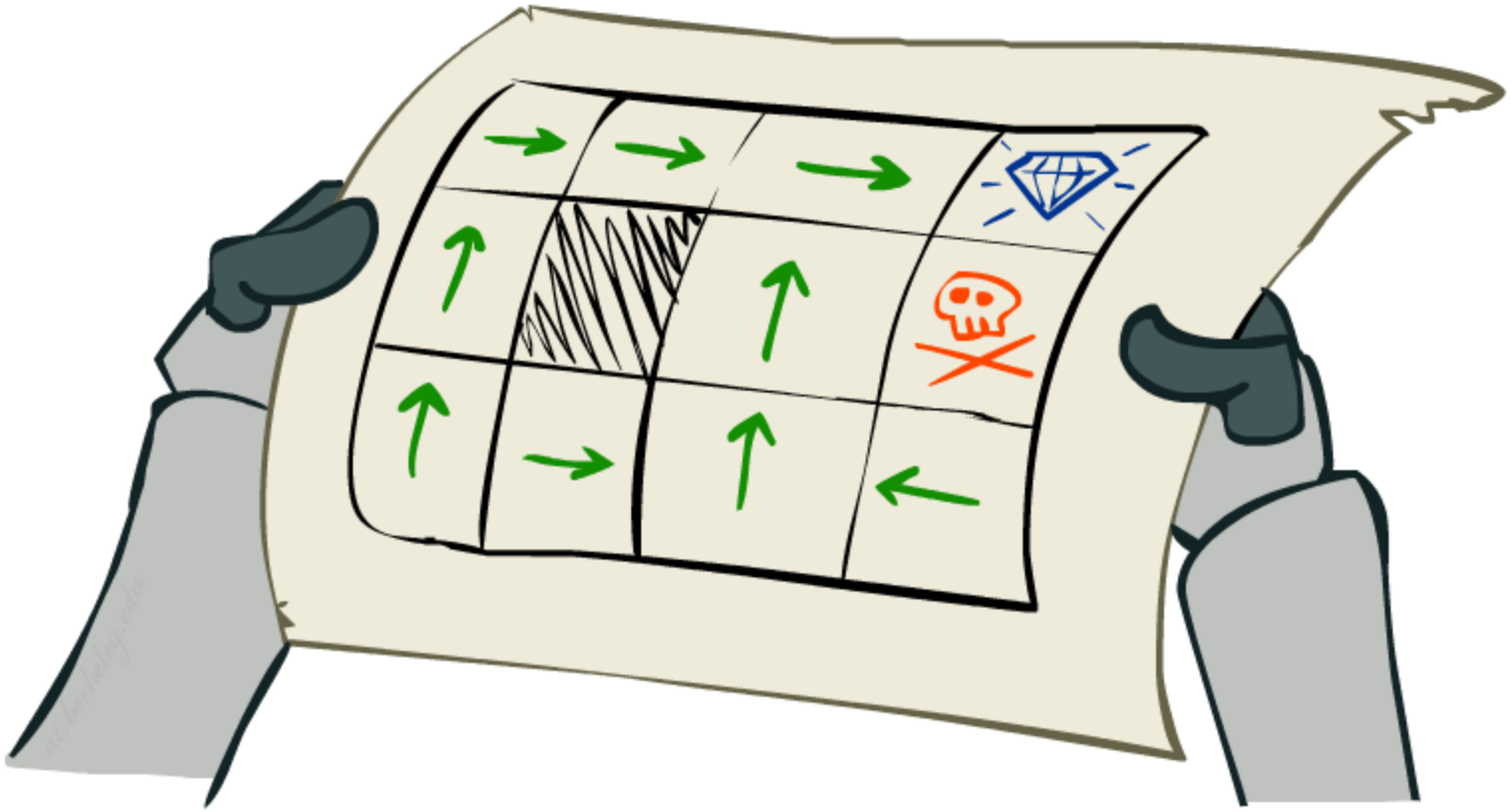
Recap: Defining MDPs

- Markov decision processes:
  - Set of states **S**
  - Start state s_0
  - Set of actions ***A***
  - Transitions ***P(s'|s, a)*** (or ***T(s, a, s')***)
  - Rewards ***R(s, a, s')*** (and discount γ)



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

Solving MDPs



Optimal Quantities

- The value (utility) of a state **s**:

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

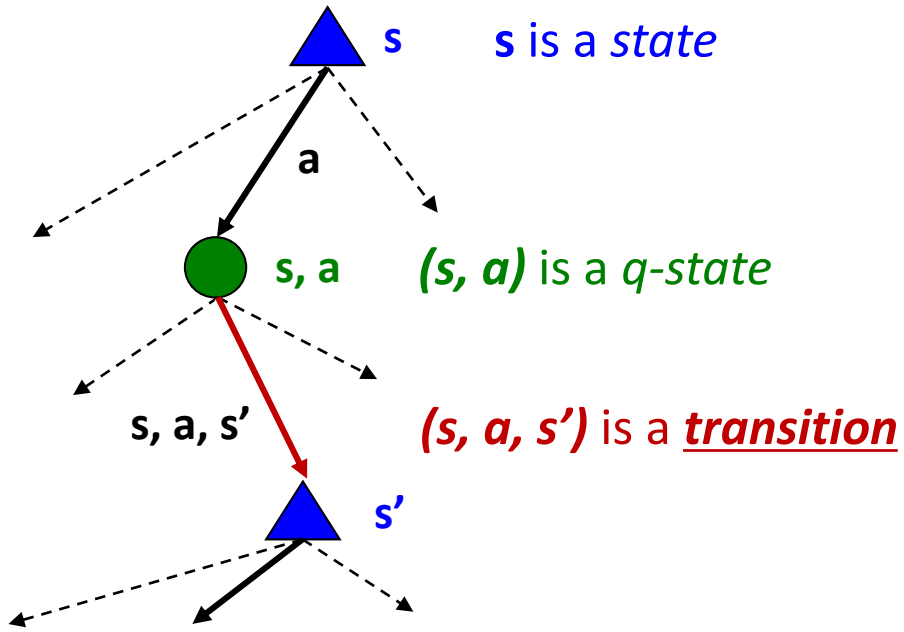
- The value (utility) of a **q-state (s, a)**:

  $Q^*(s, a)$ = Expected utility starting out having taken action ***a*** from state ***s*** and **(thereafter) acting optimally**

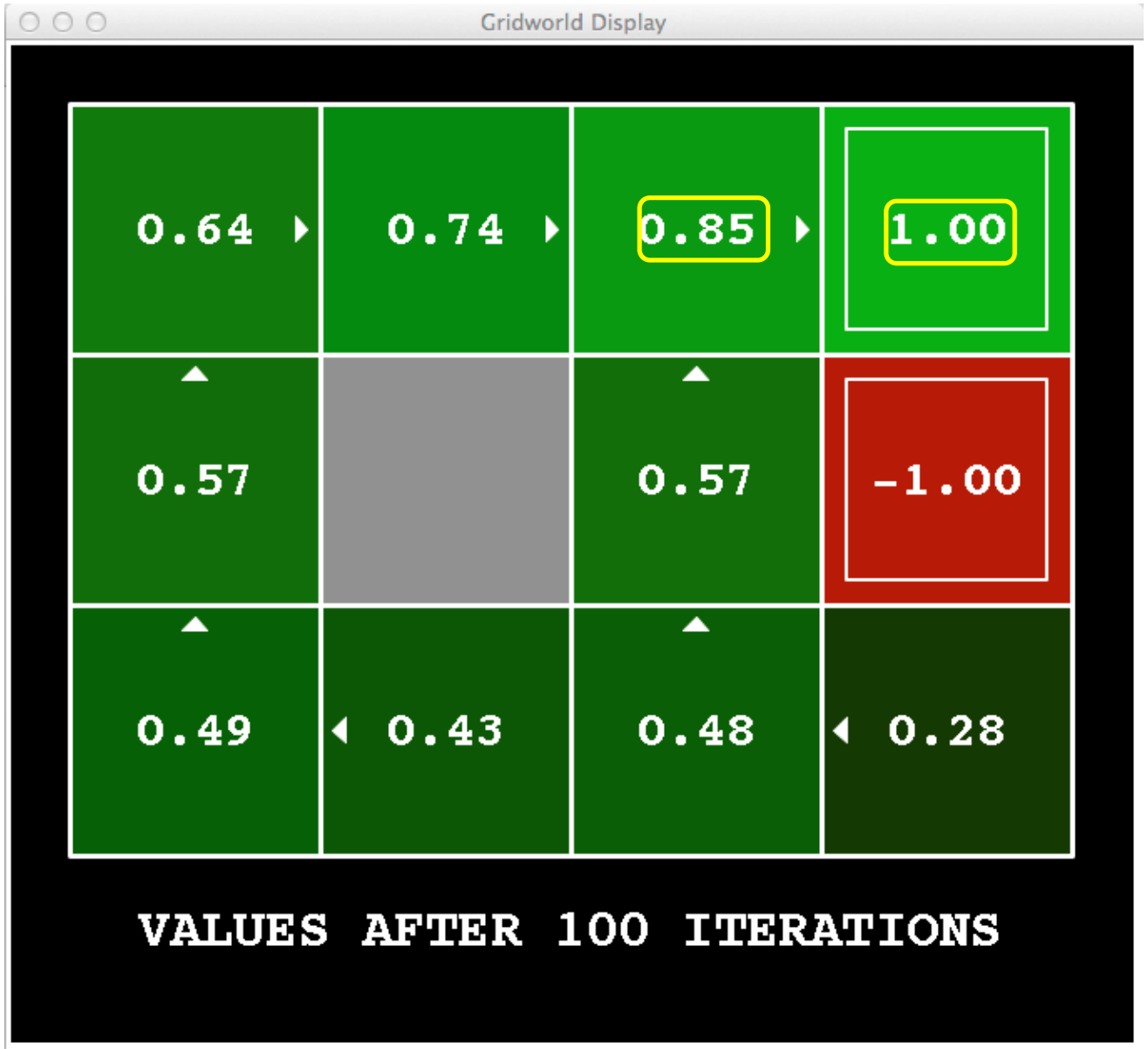
- The optimal policy:

  $\pi^*(s)$ = Optimal action from state **s**

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

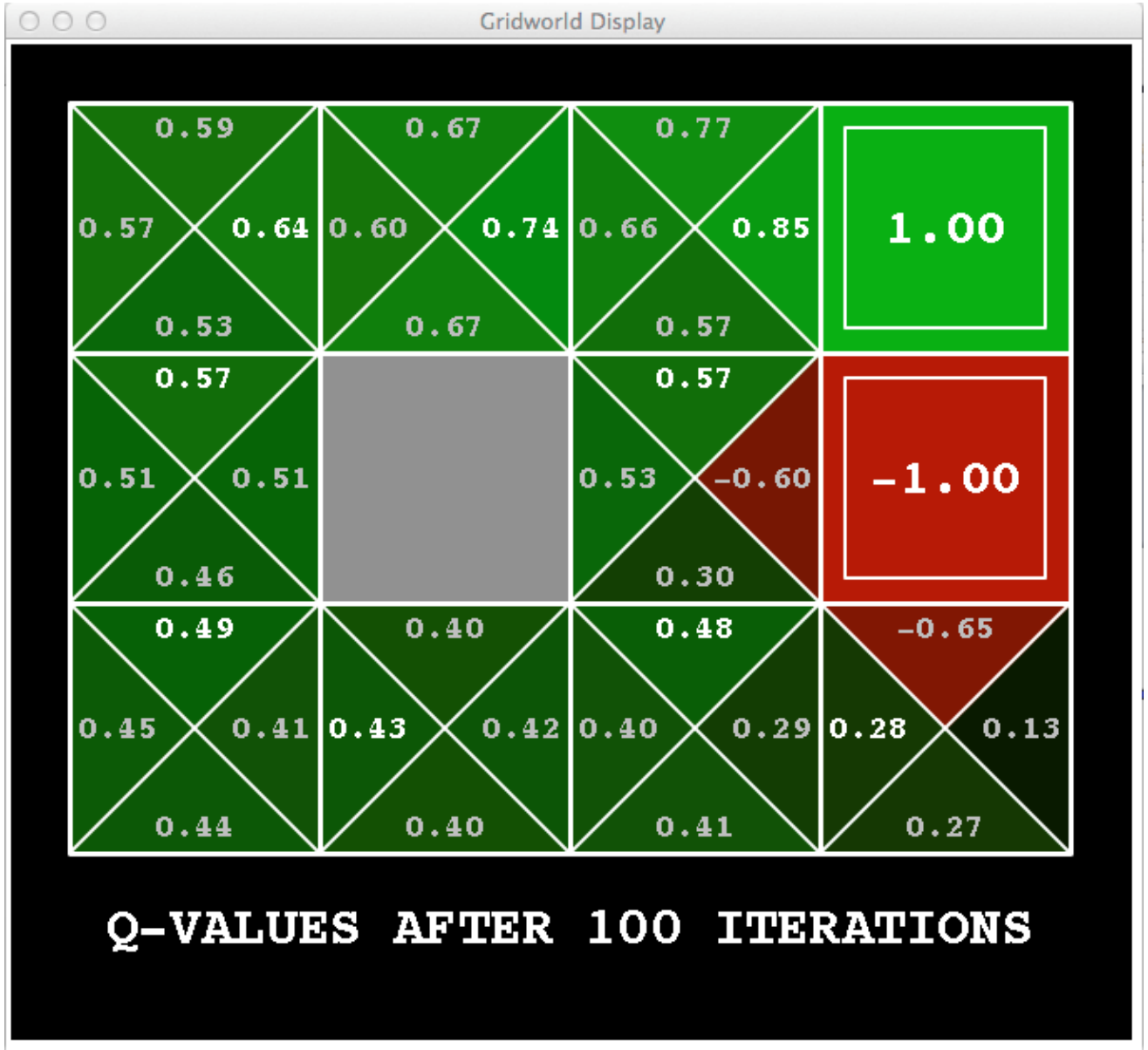


Gridworld *V* Values (Cont'd)



Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

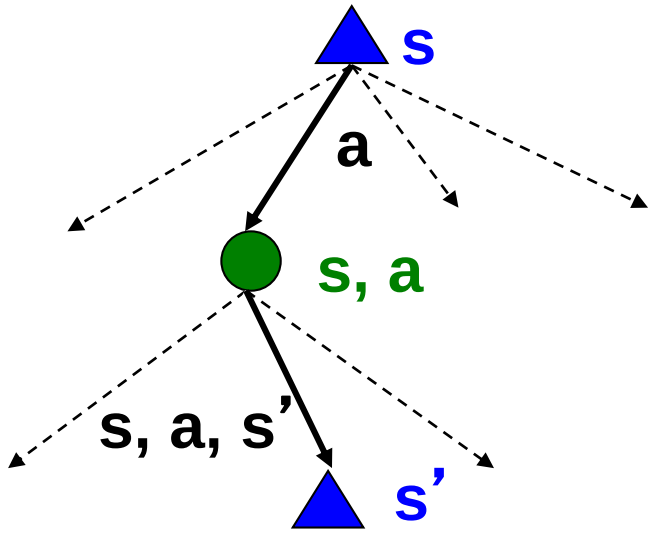
Gridworld *Q* Values (Cont'd)



Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Values of States

- Fundamental operation: Compute the (***expectimax***) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards



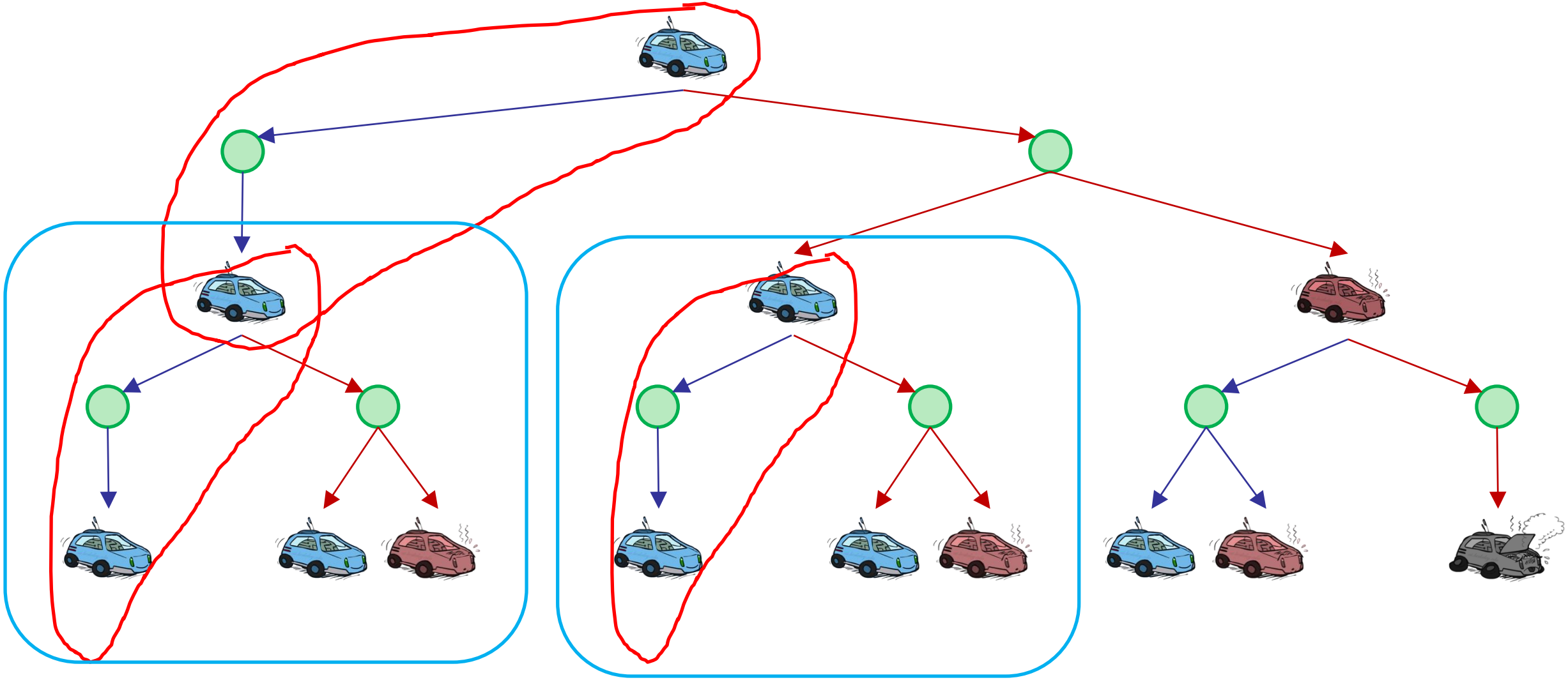
- Recursive definition of value **V**:

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

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

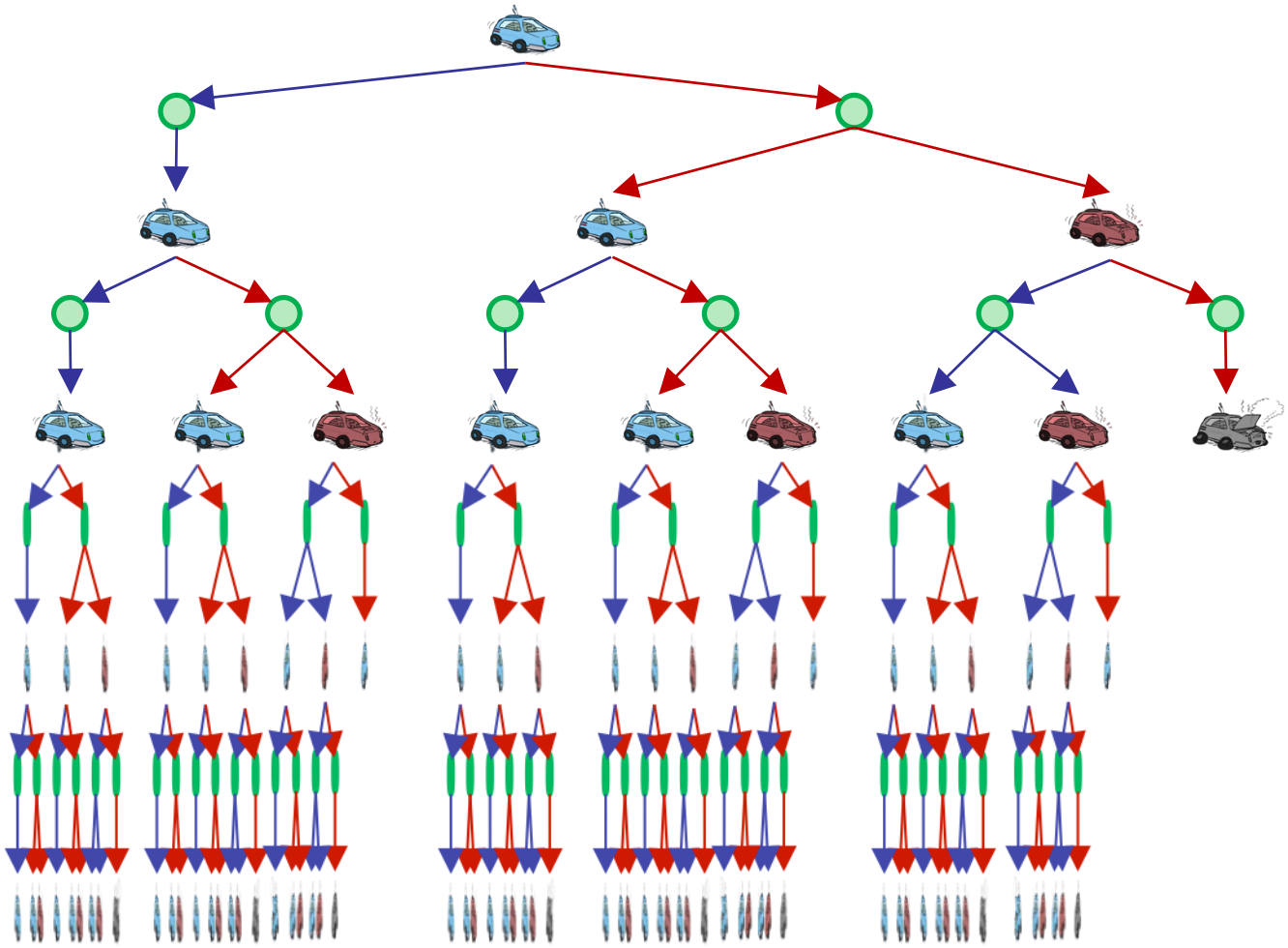
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

Racing Search Tree



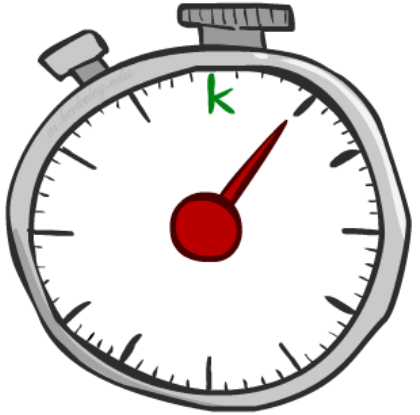
Racing Search Tree

- We are doing way too much work with expectimax!
- Problem: States are repeated
  - **Idea:** Only compute needed quantities once
- Problem: Tree goes on forever
  - **Idea:** Do a depth-limited computation, but with increasing depths until change is small
  - **Note:** Deep parts of the tree eventually do not matter if $\gamma < 1$

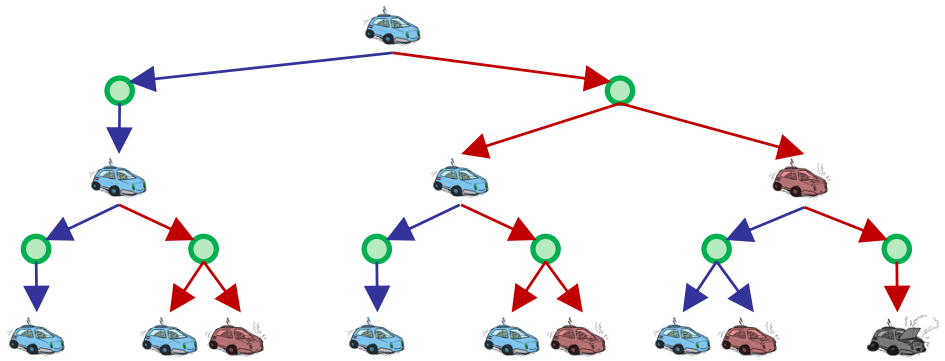


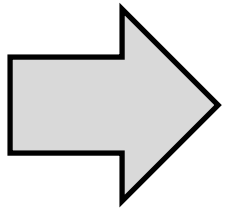
Time-Limited Values

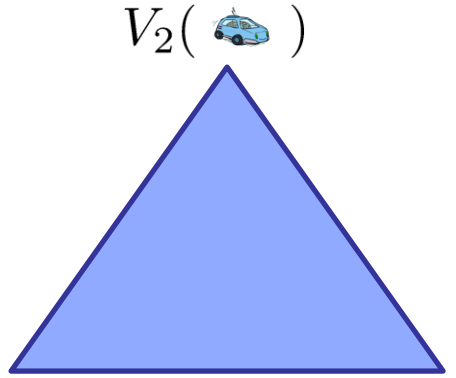
- **Key idea:** Time-limited values



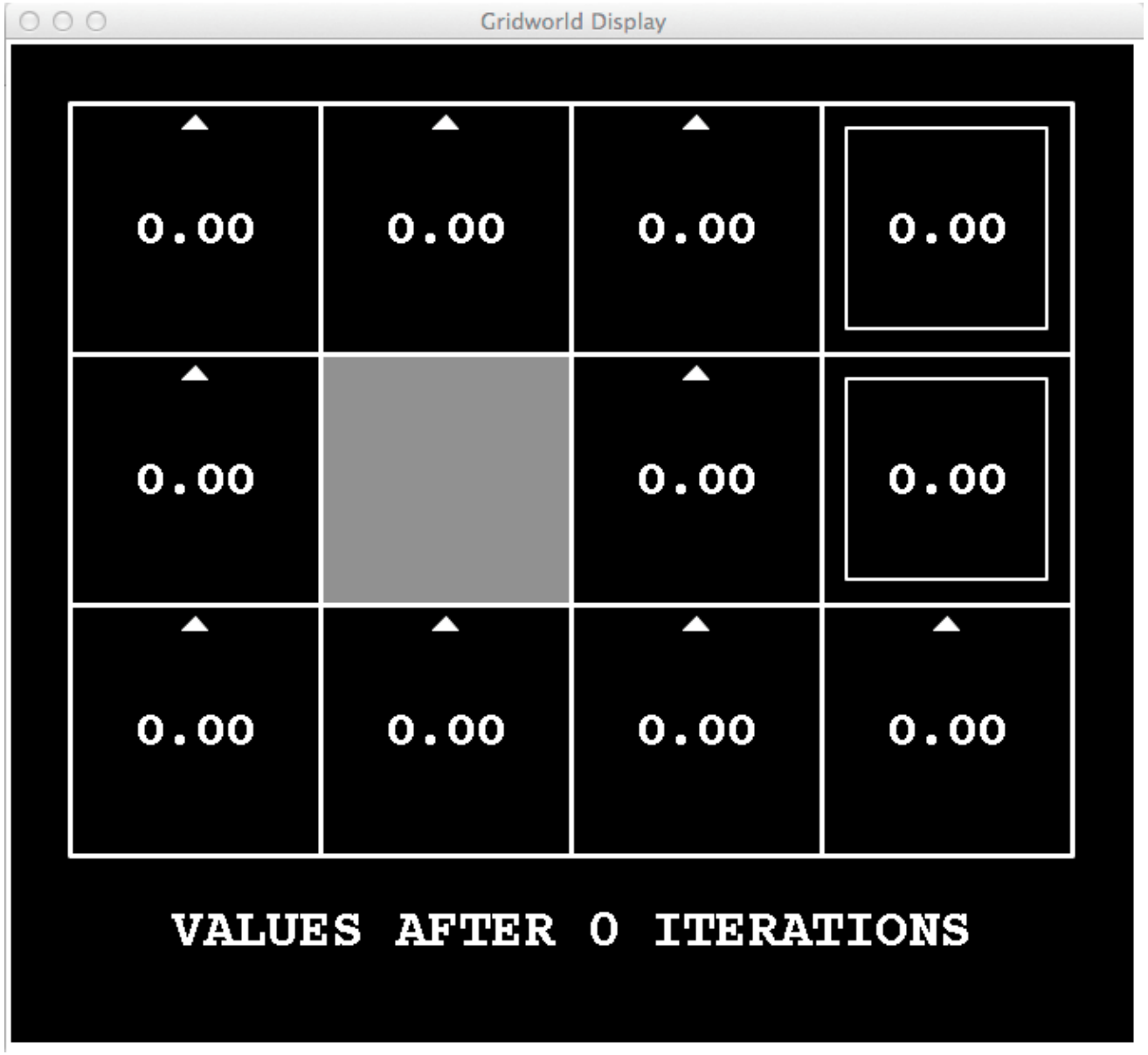
- Define $V_k(s)$ to be the optimal value of **s** if the game ends in ***k*** more time steps
  - Equivalently, it is what a depth-k expectimax would give from **s**





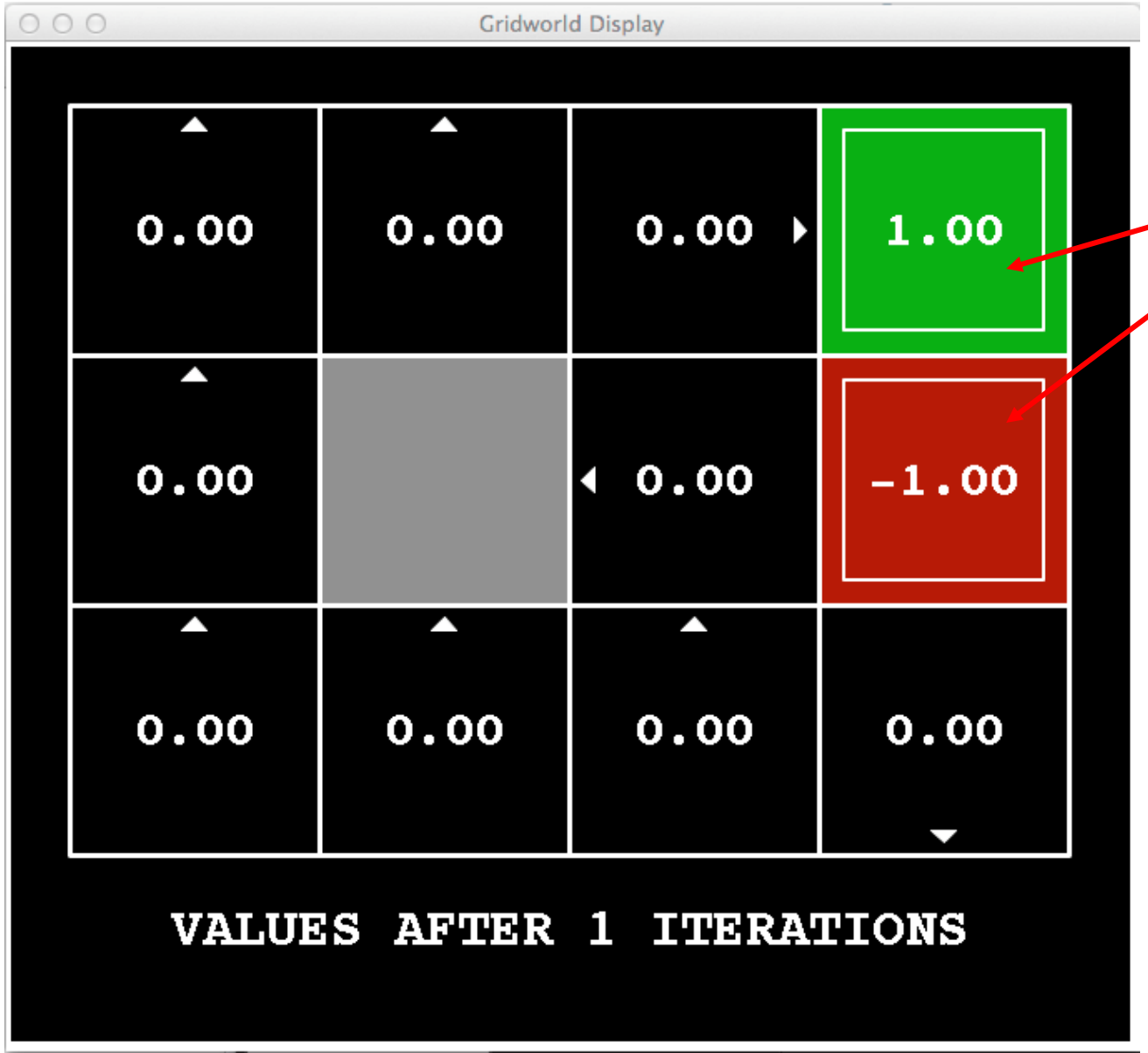


Iteration $k = 0$



Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 1*



One step required to get rewards

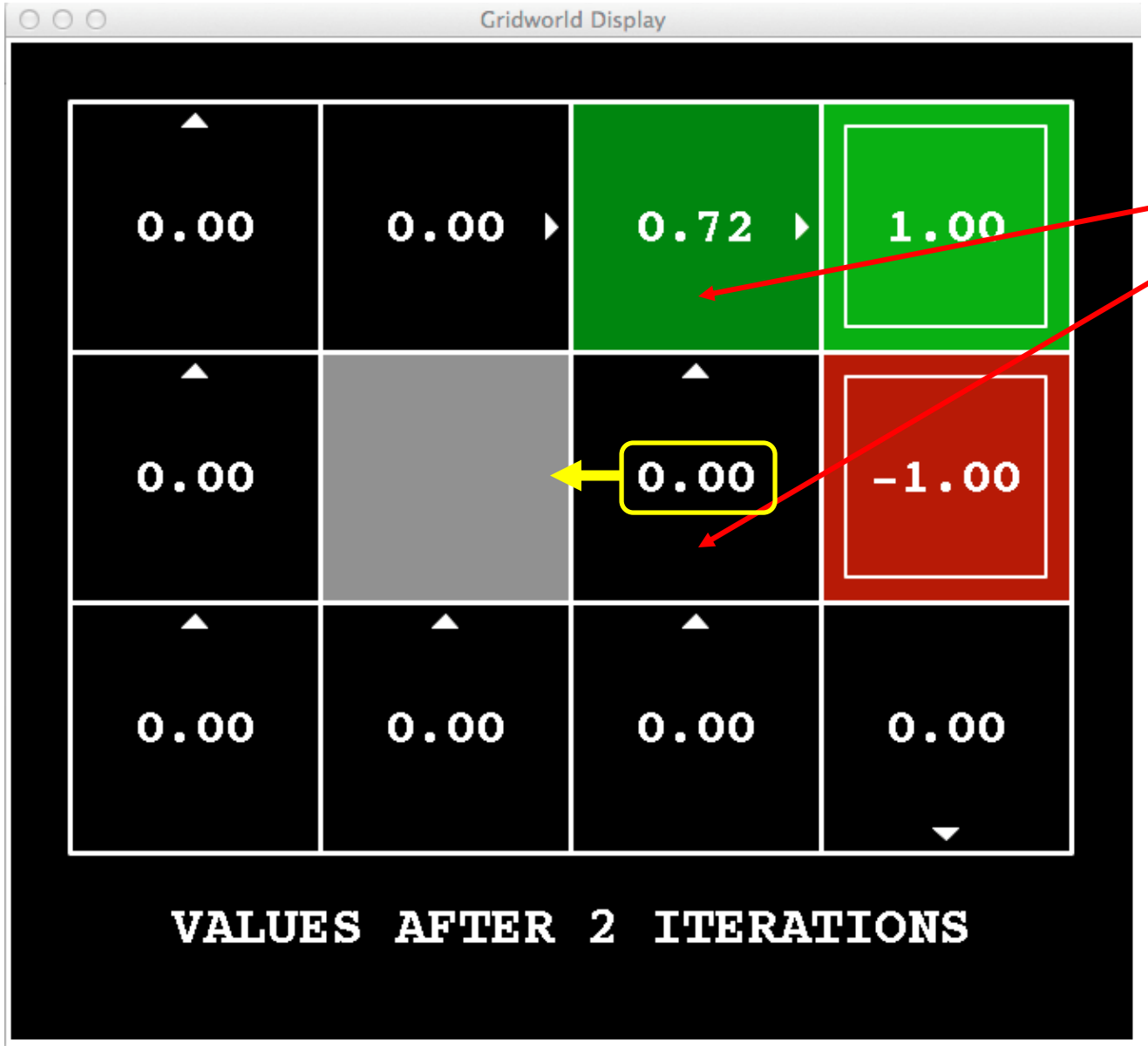
Exit action

Noise = ***0.2***

Discount: (γ = ***0.9***)

Living reward: (***r = 0***)

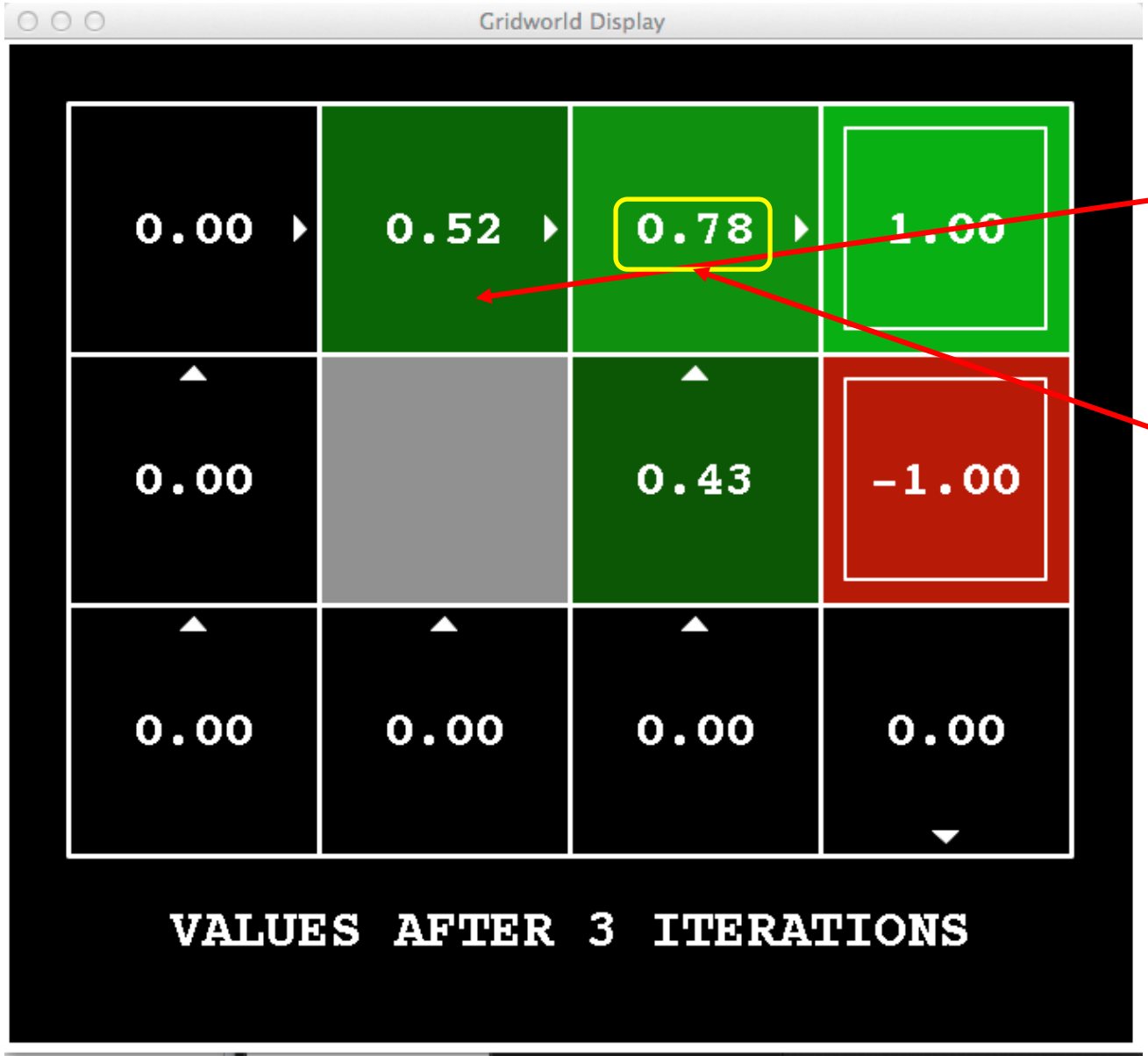
Iteration *k = 2*



Two steps required to get rewards

Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 3*

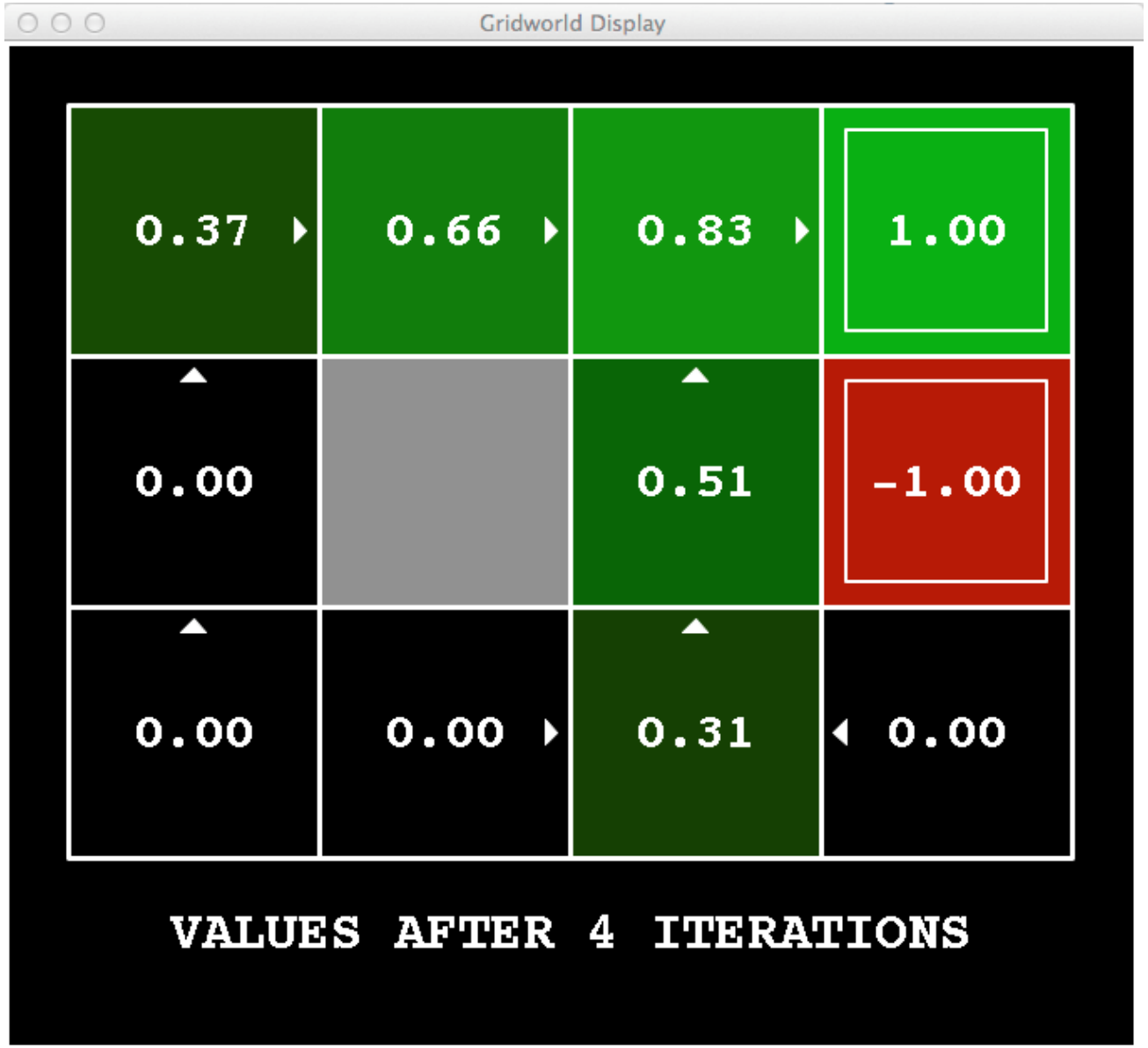


Three steps required to get rewards

@ k = 2

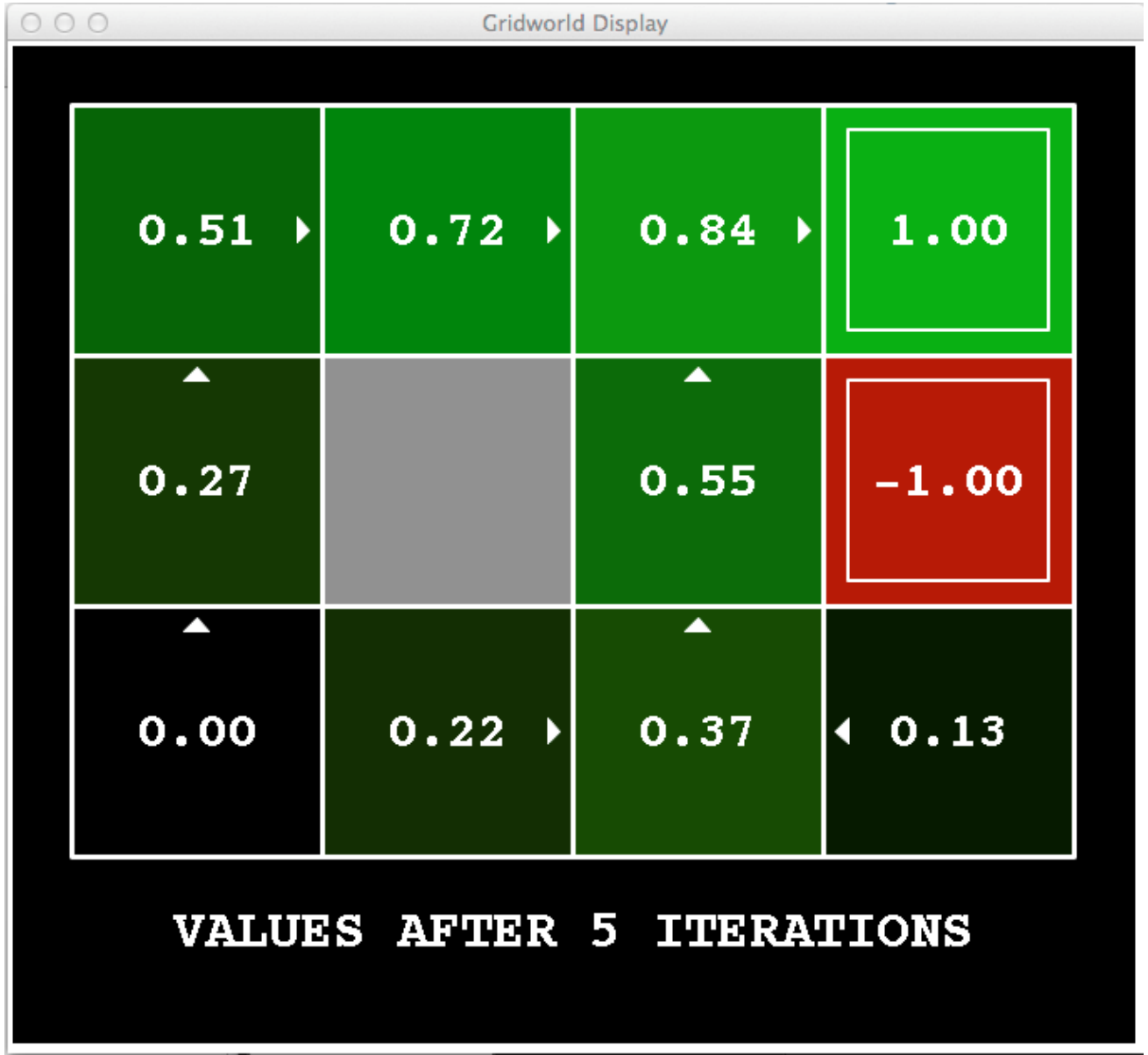
Noise = ***0.2***
Discount: (***γ = 0.9***)
Living reward: (***r = 0***)

Iteration *k = 4*



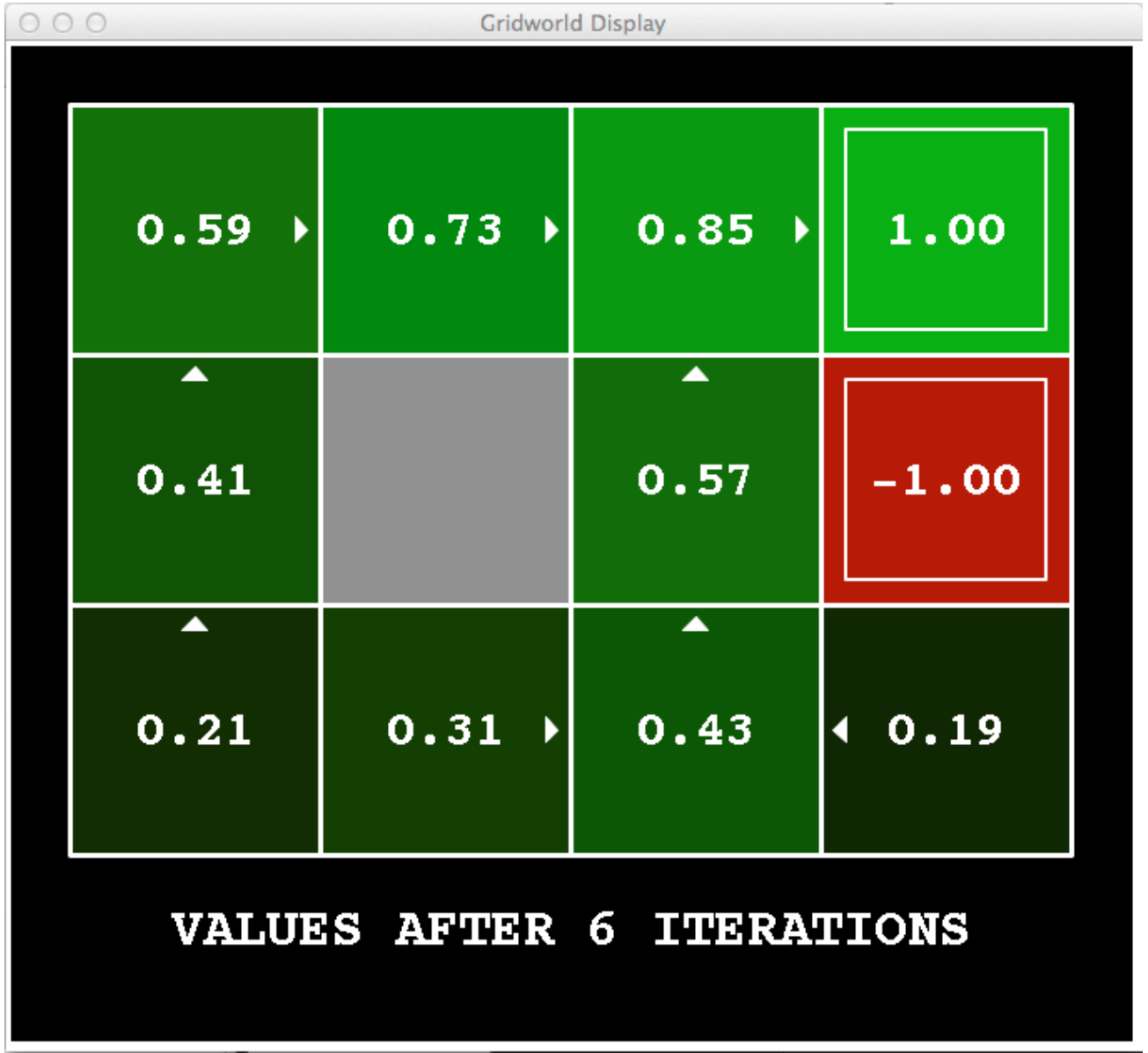
Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 5*



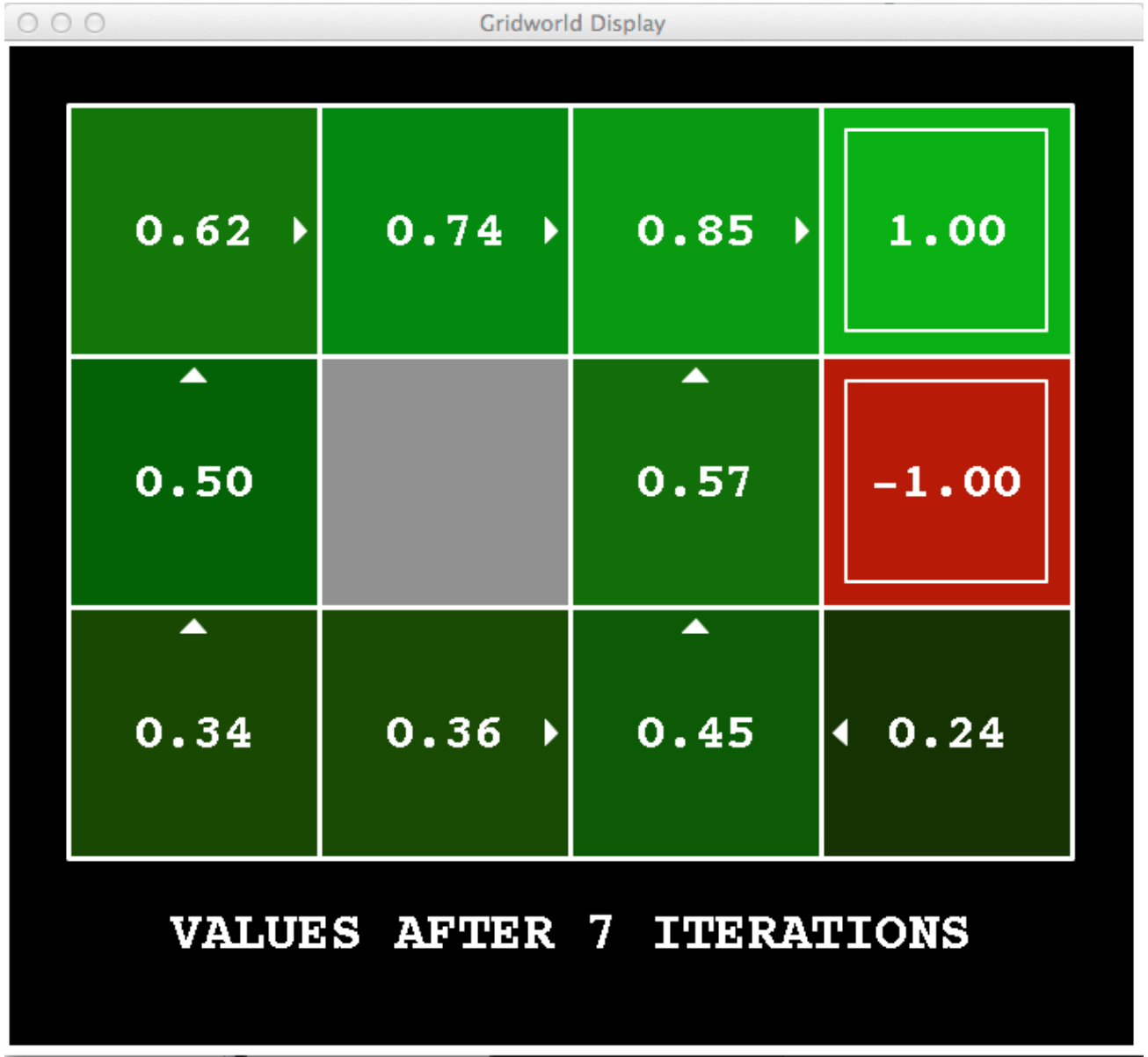
Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 6*



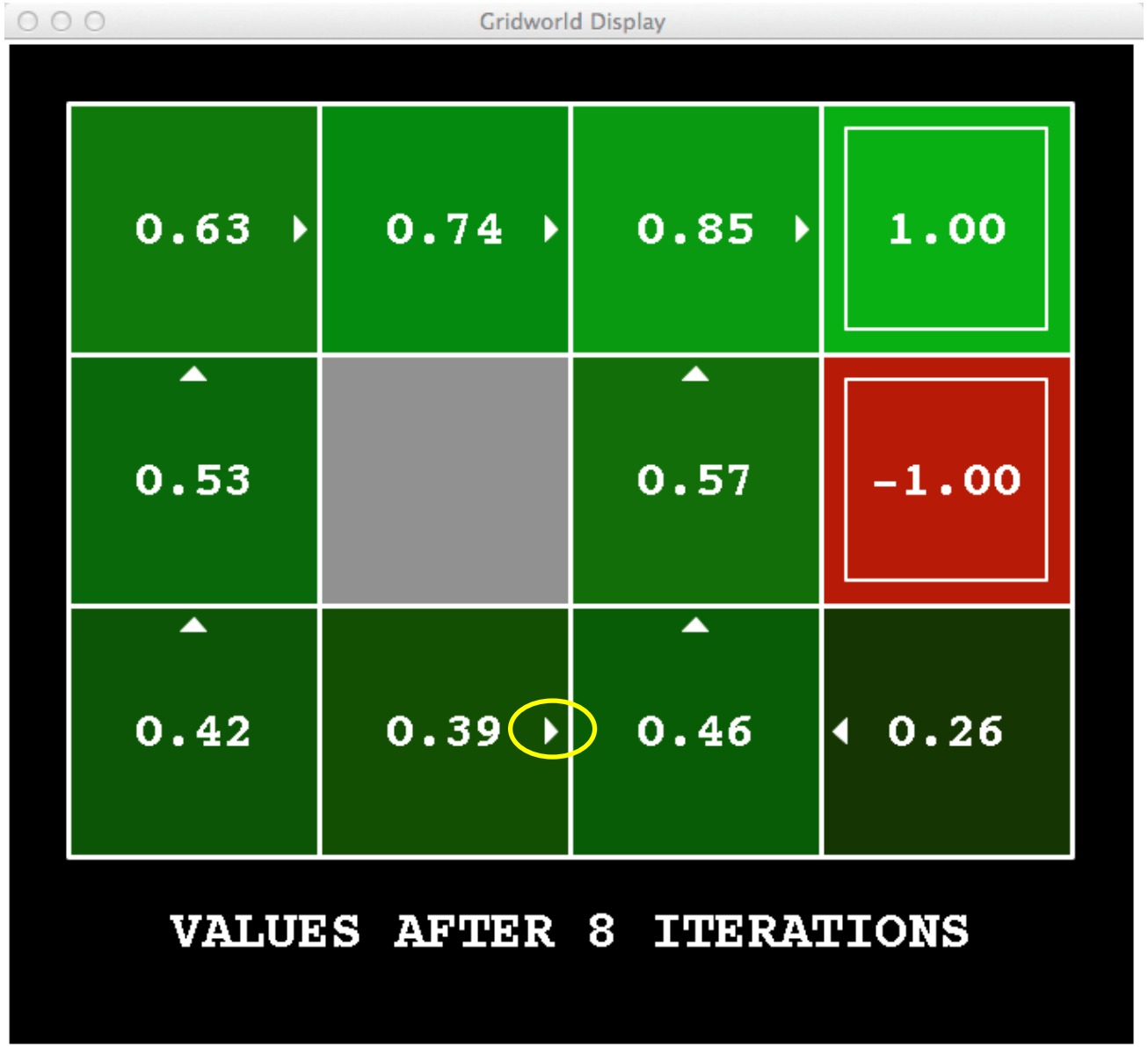
Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 7*



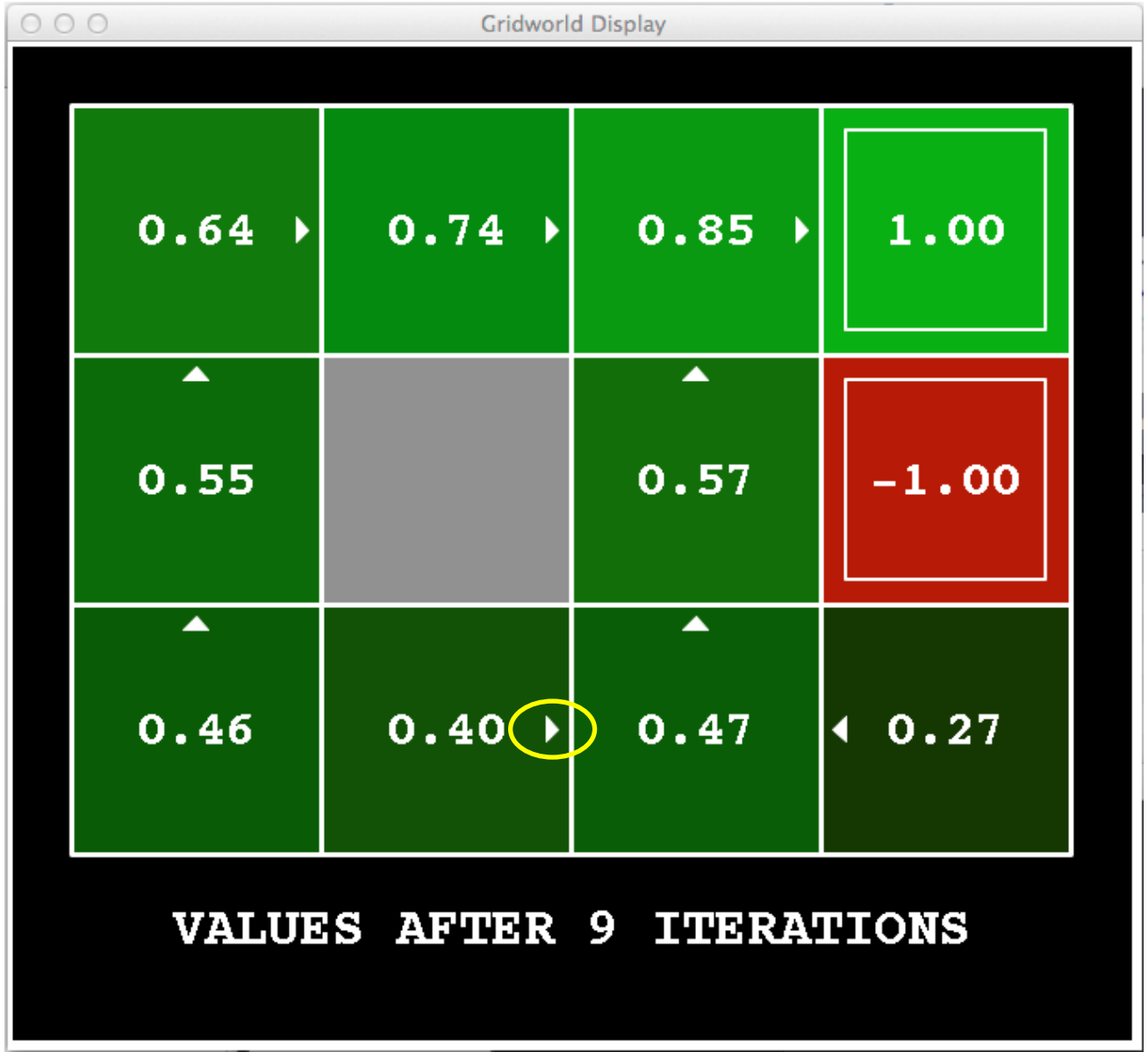
Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 8*



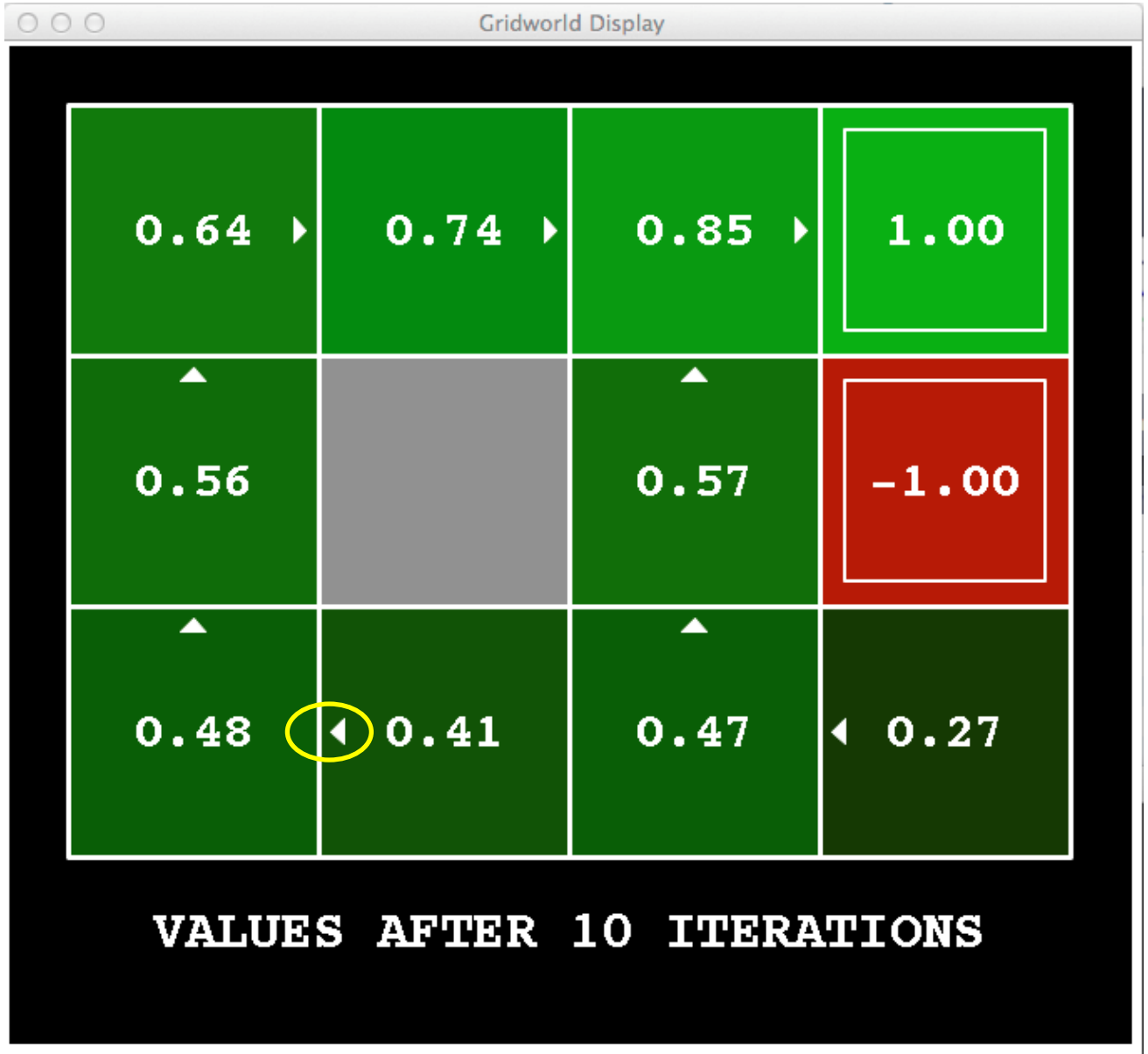
Noise = ***0.2***
Discount: (***γ = 0.9***)
Living reward: (***r = 0***)

Iteration *k = 9*



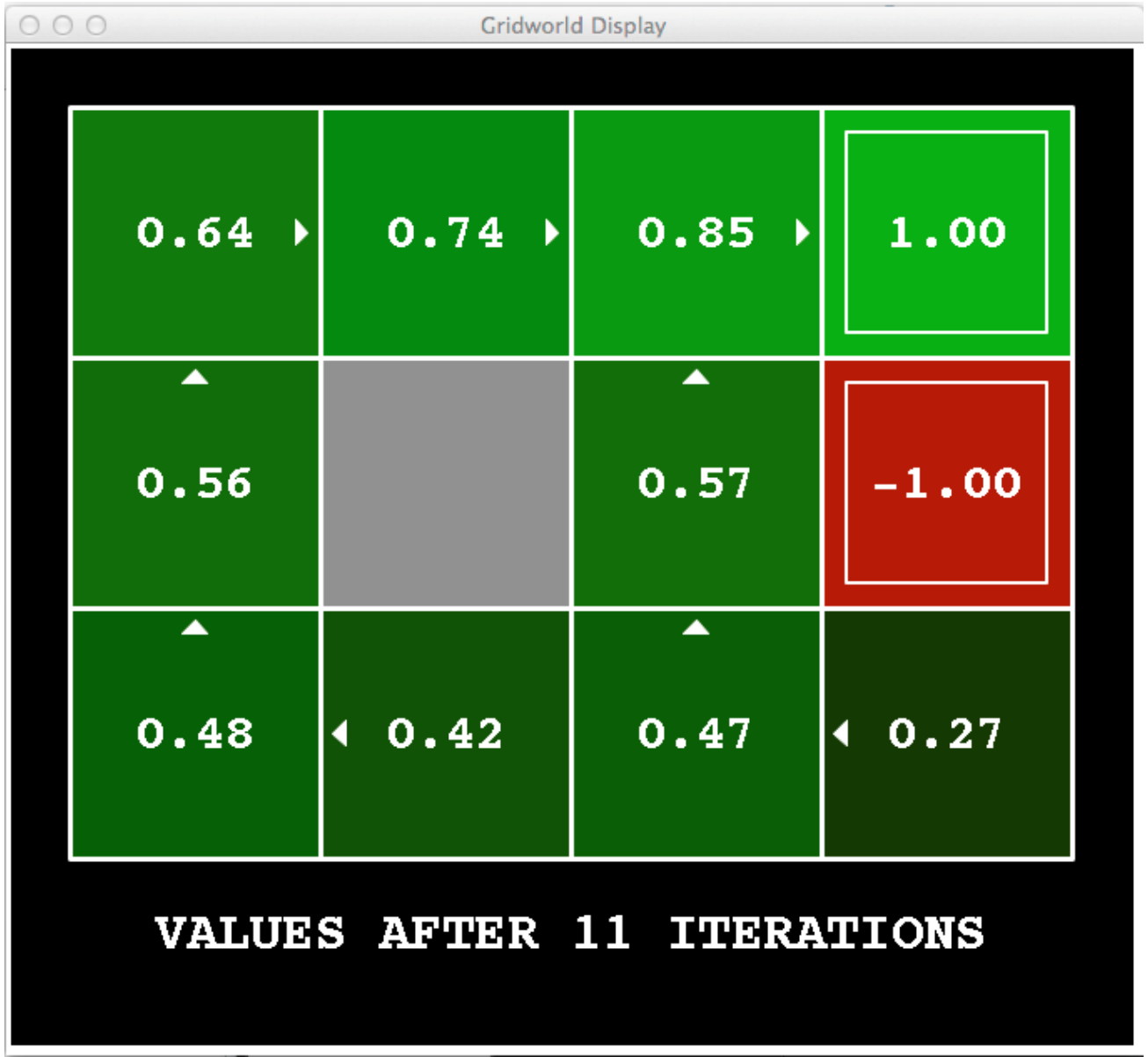
Noise = ***0.2***
Discount: (***γ = 0.9***)
Living reward: (***r = 0***)

Iteration *k = 10*



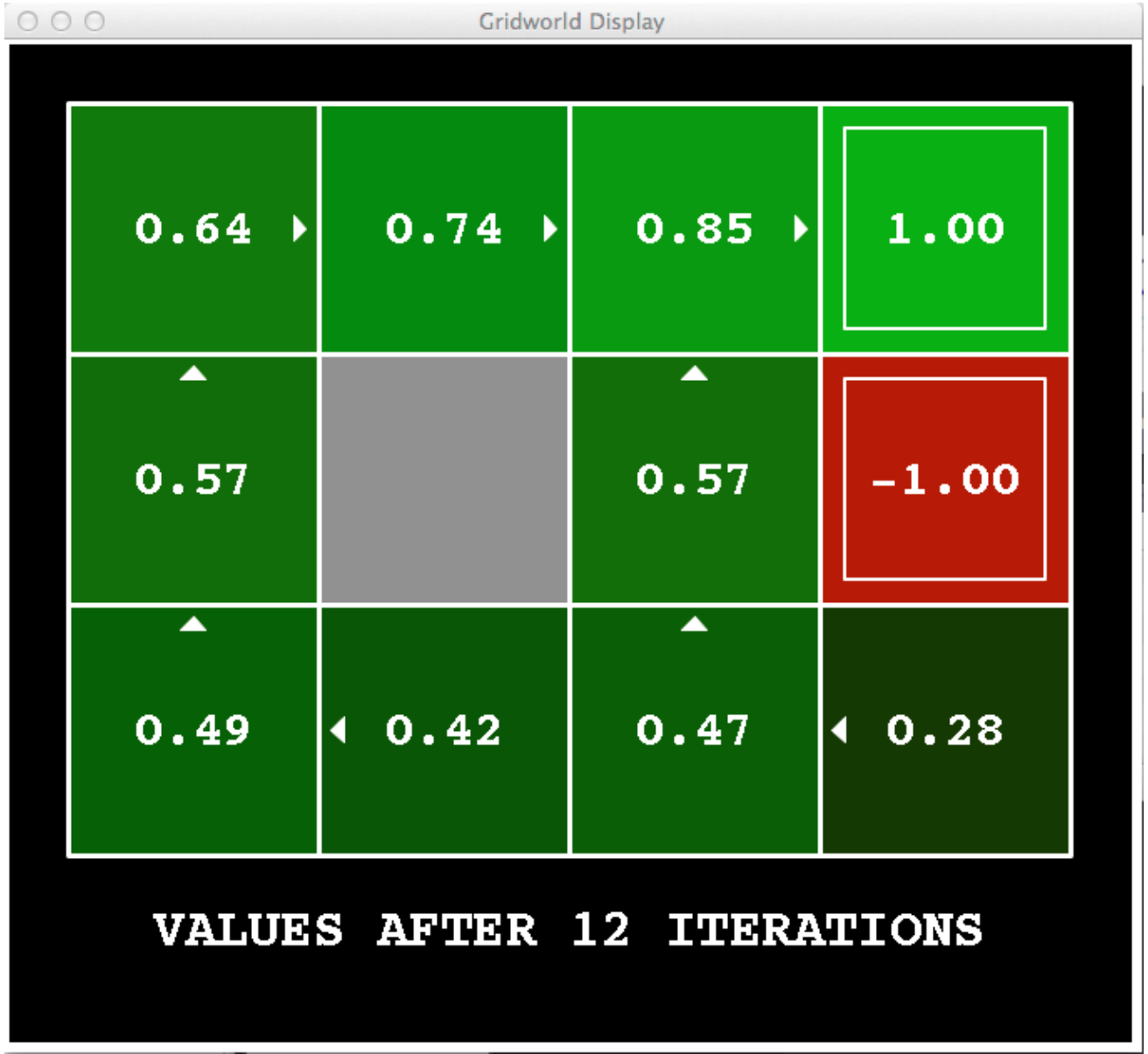
Noise = ***0.2***
Discount: (***γ = 0.9***)
Living reward: (***r = 0***)

Iteration *k = 11*



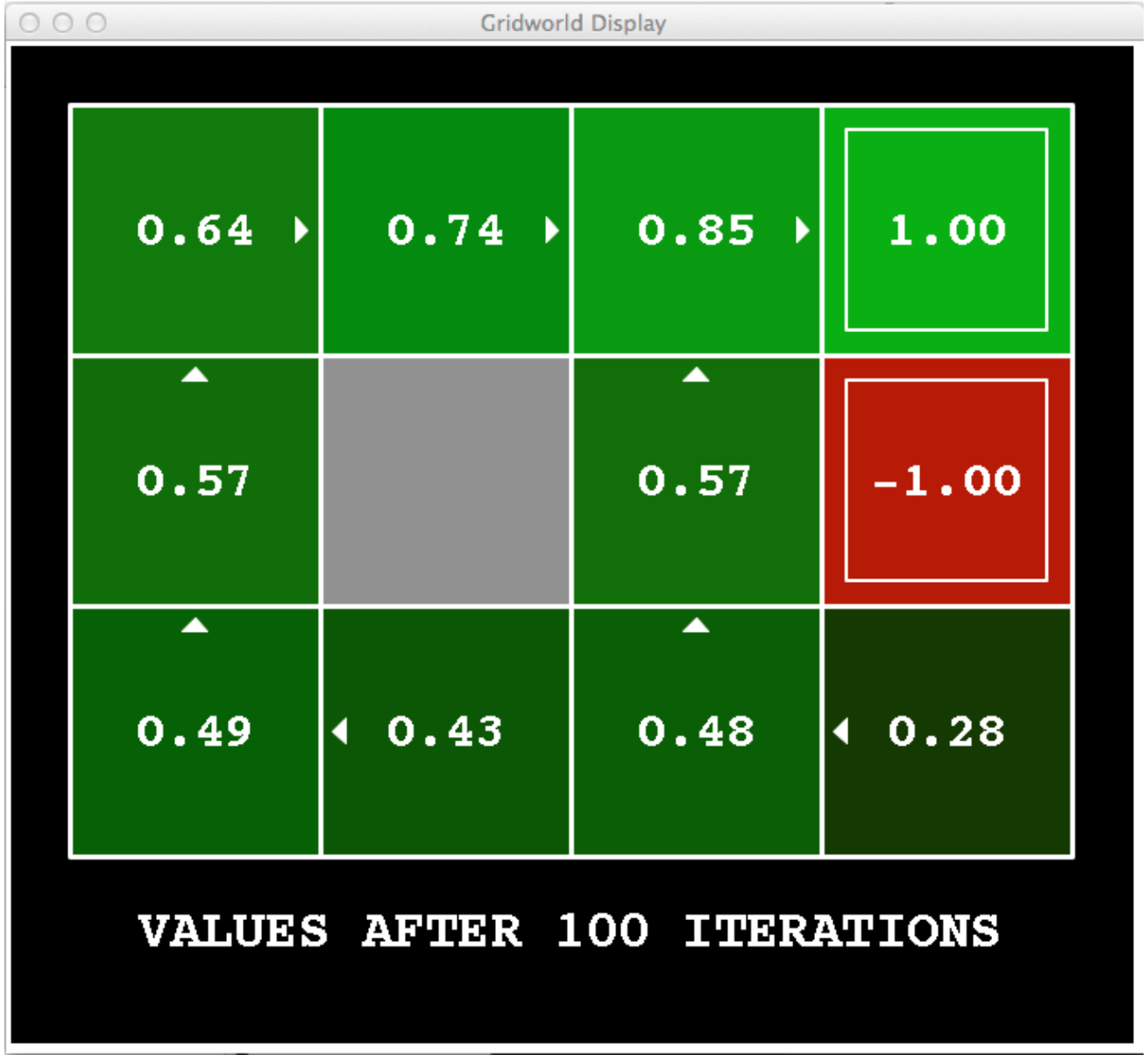
Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 12*



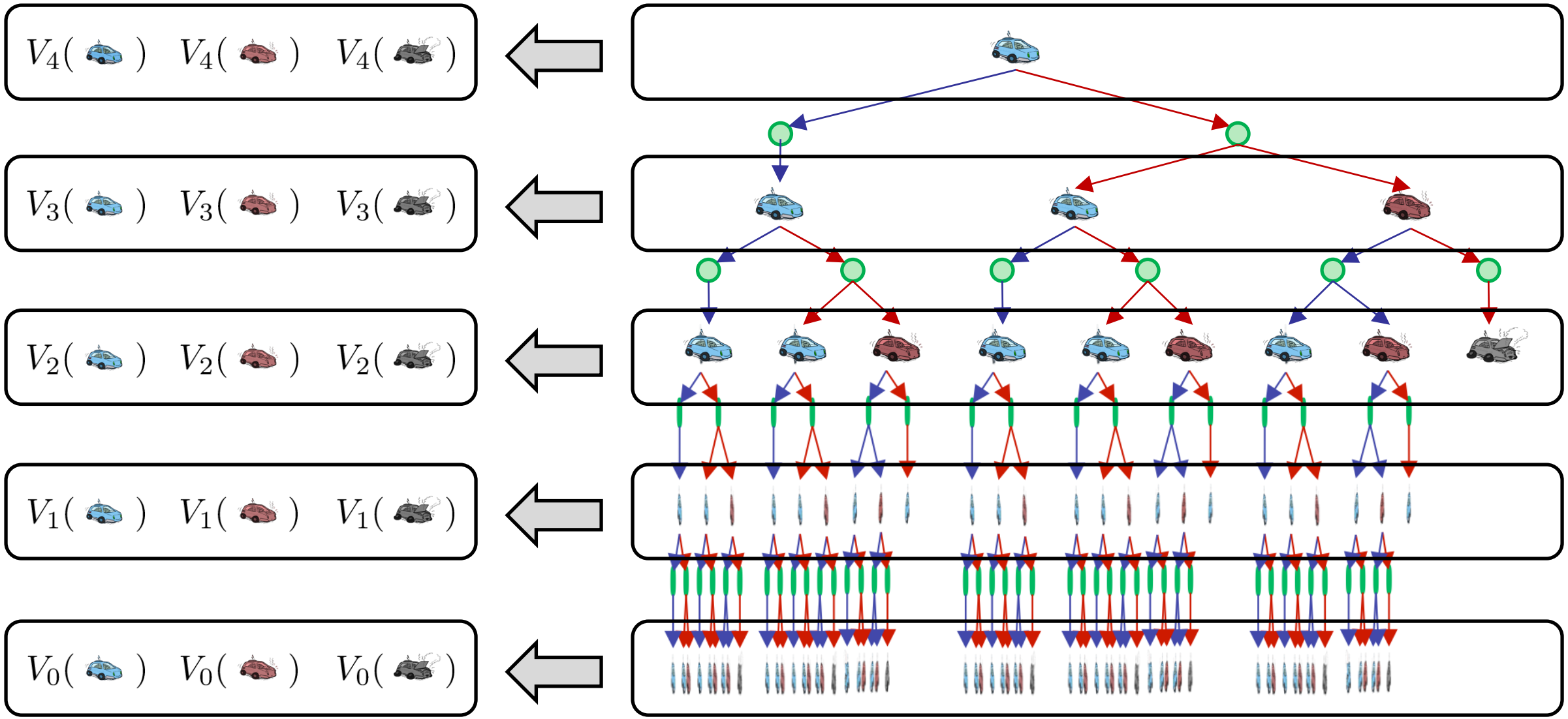
Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Iteration *k = 100*

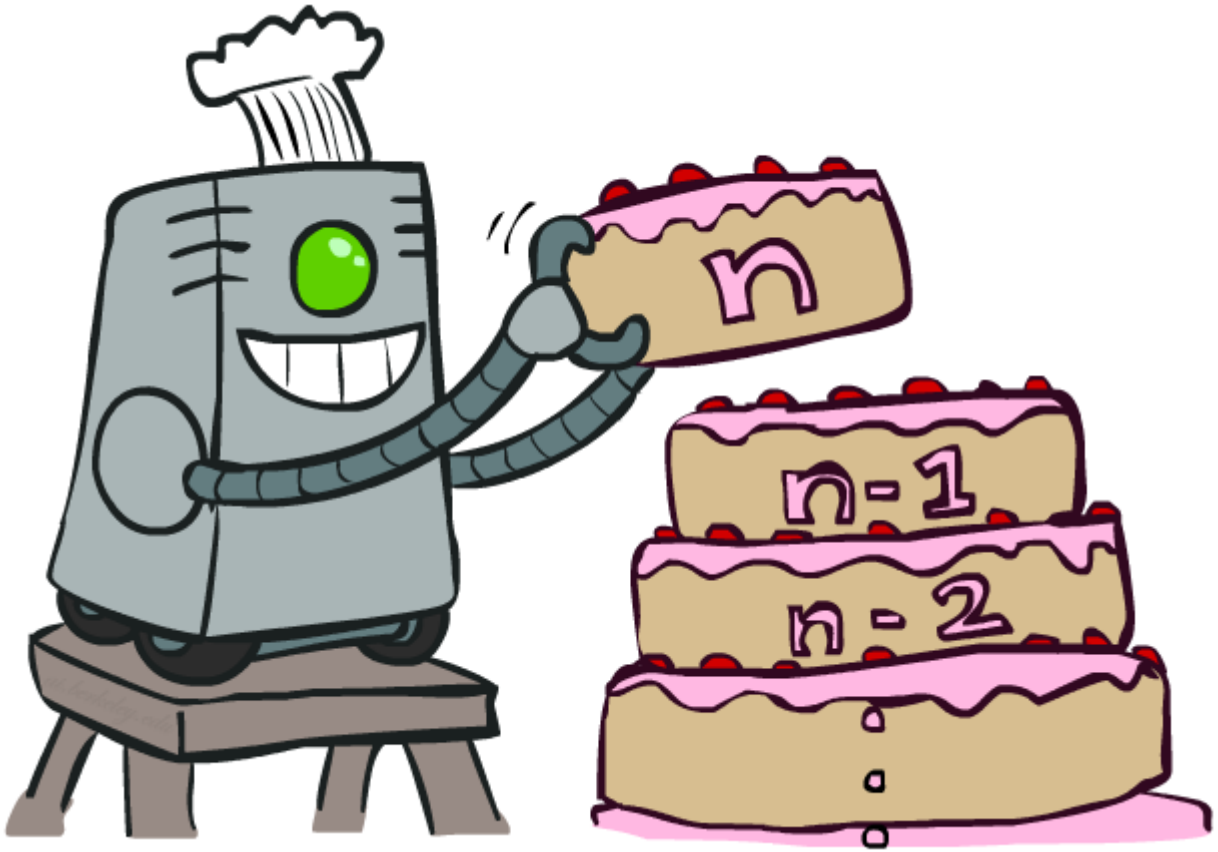


Noise = ***0.2***
Discount: (γ = ***0.9***)
Living reward: (***r = 0***)

Computing Time-Limited Values



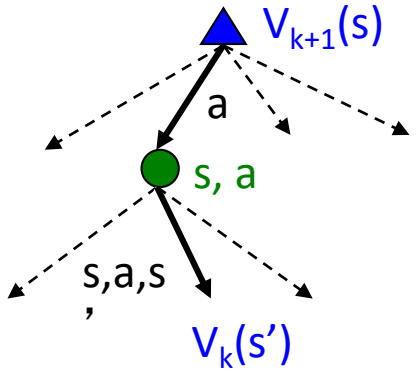
Value Iteration



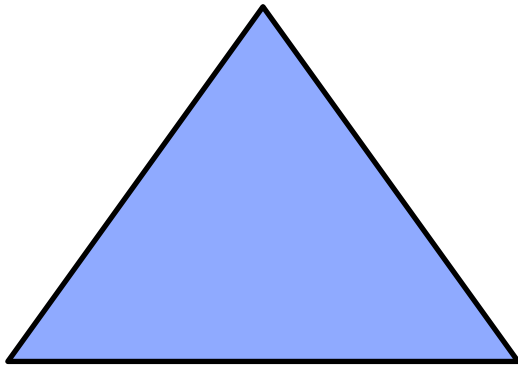
Value Iteration

- Start with $V_0(s) = 0$: No time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

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



- Repeat until **convergence** ($||V_{k+1}(s) - V_k(s)|| \leq \varepsilon$)
- Complexity of each iteration: $O(S^2A)$ where we have **S** possible states and **A** actions available



Value Iteration (Cont'd)

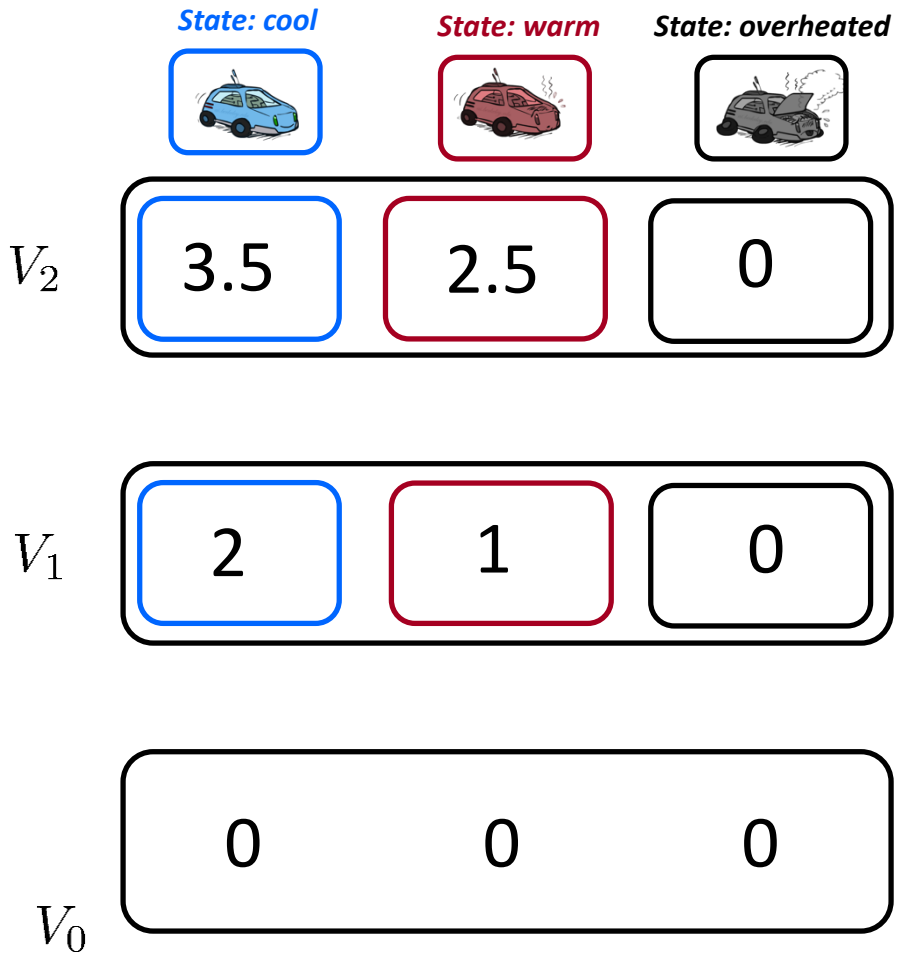
- In case we have:

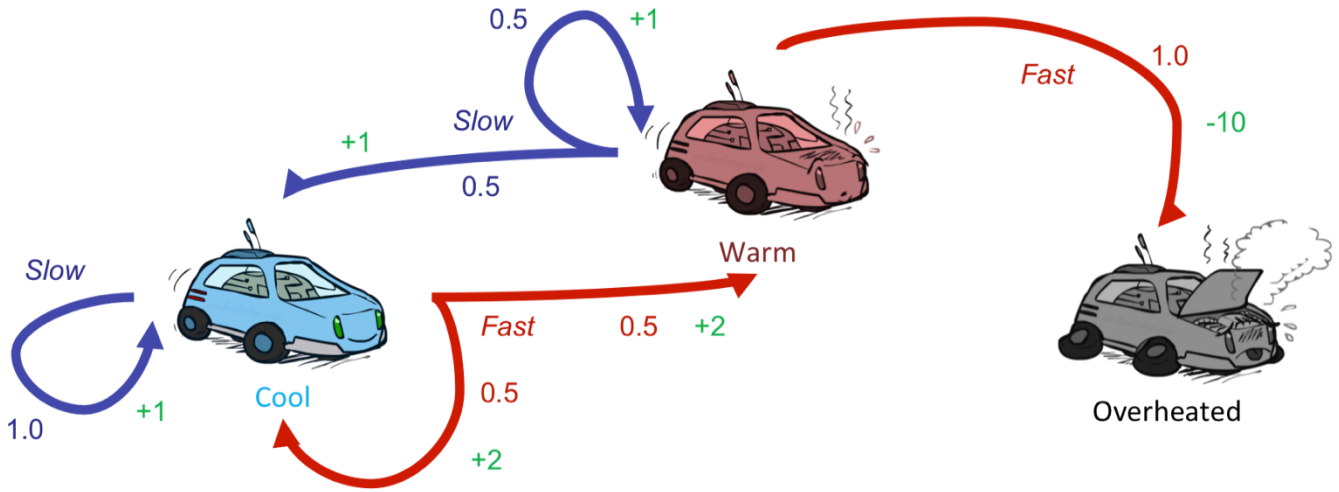
$$R(s, a, s') = R(s, a)$$

- Then, the value iteration update becomes:

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

Example: Value Iteration

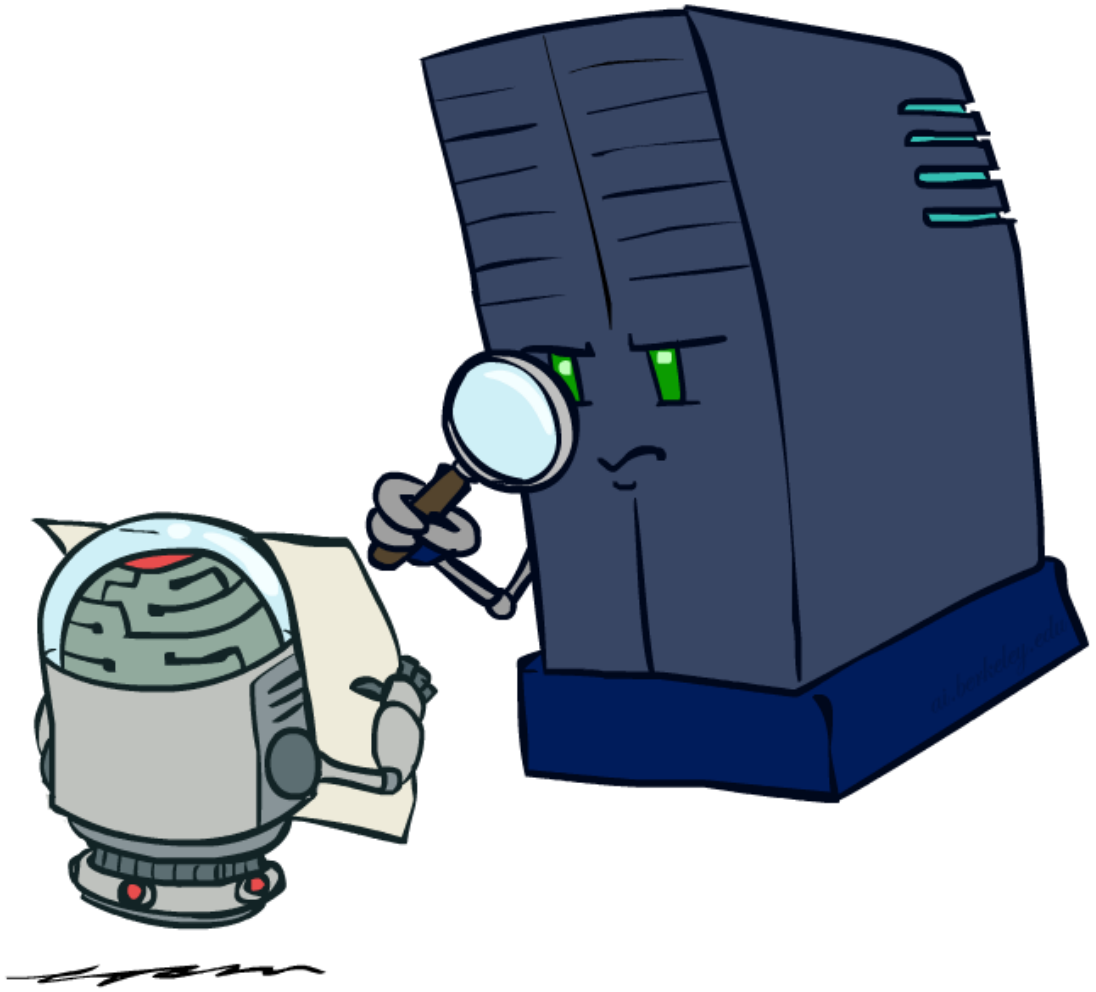




*Assume no discount (**γ = 1**)!*

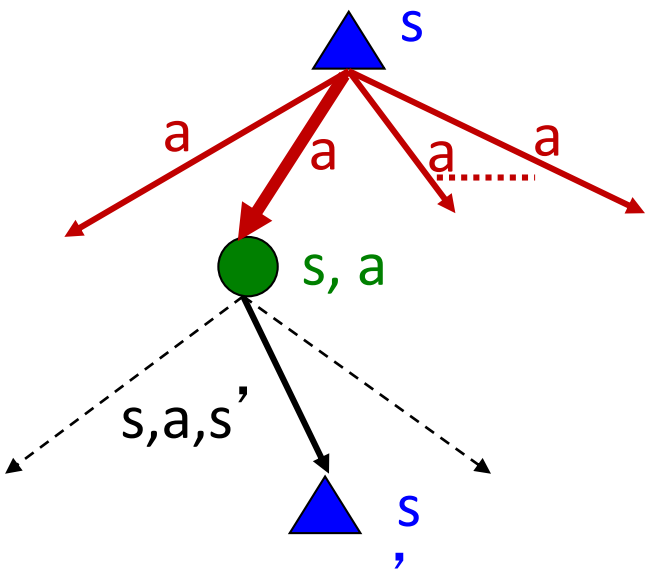
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Policy Evaluation

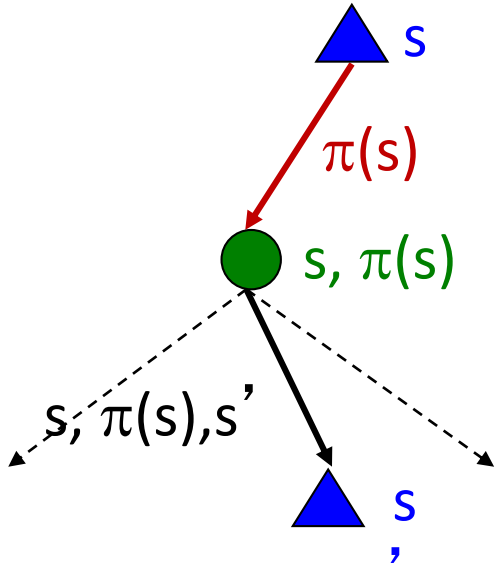


Fixed Policies

Do the optimal action



Do what π says to do



- 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

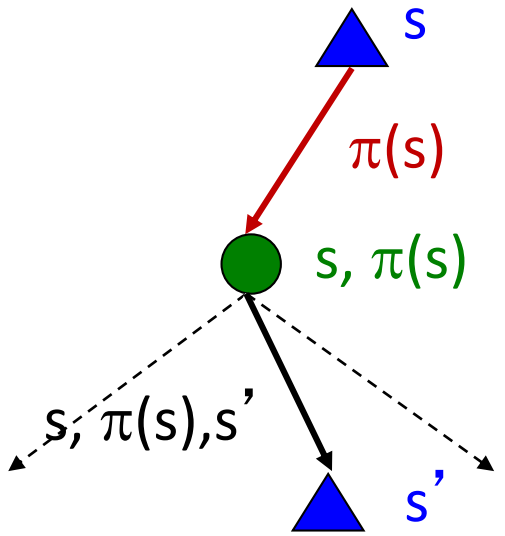
- Another basic operation: Compute the utility of a state ***s*** under a ***fixed*** (***generally non-optimal***) ***policy***

- 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

No max operator!

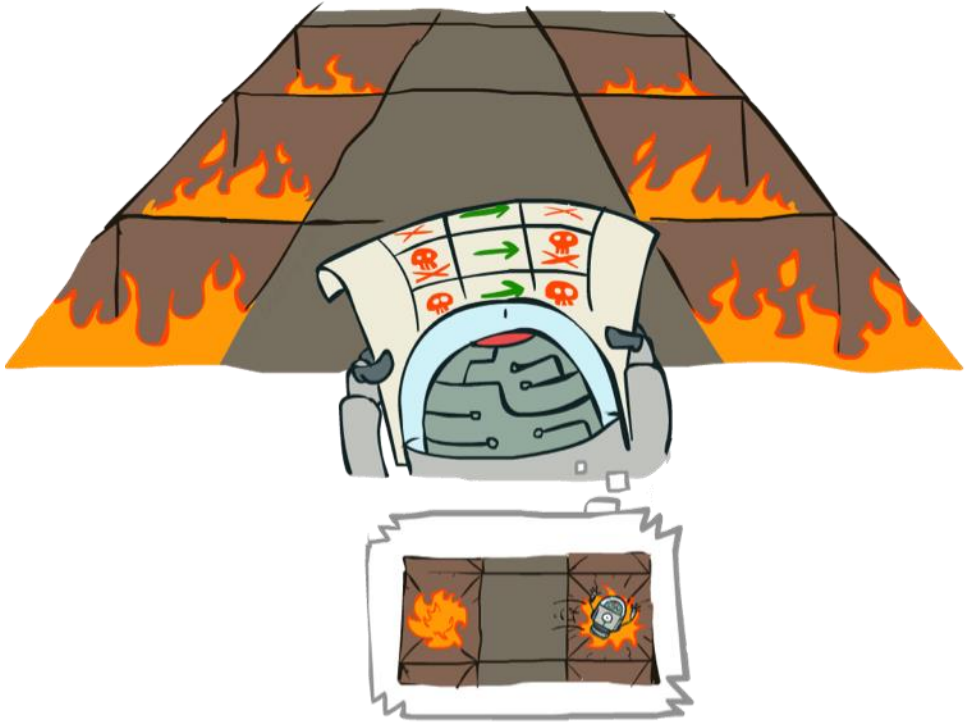
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$



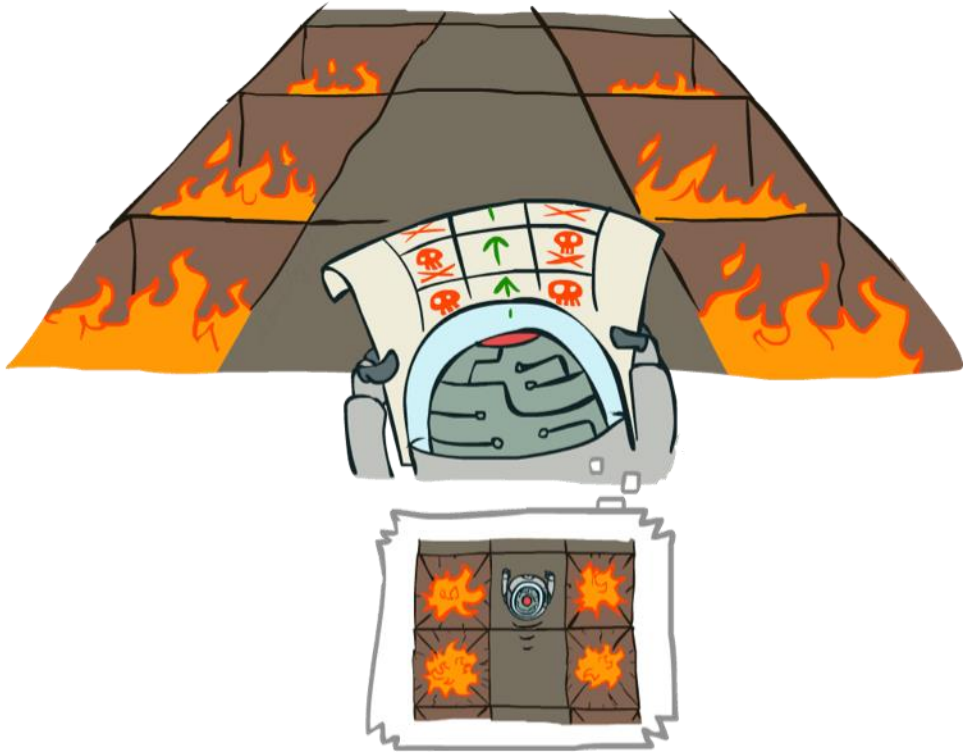
- Recursive relation (**one-step look-ahead / Bellman equation**):

Example: Policy Evaluation

Always Go Right



Always Go Forward



Example: Policy Evaluation

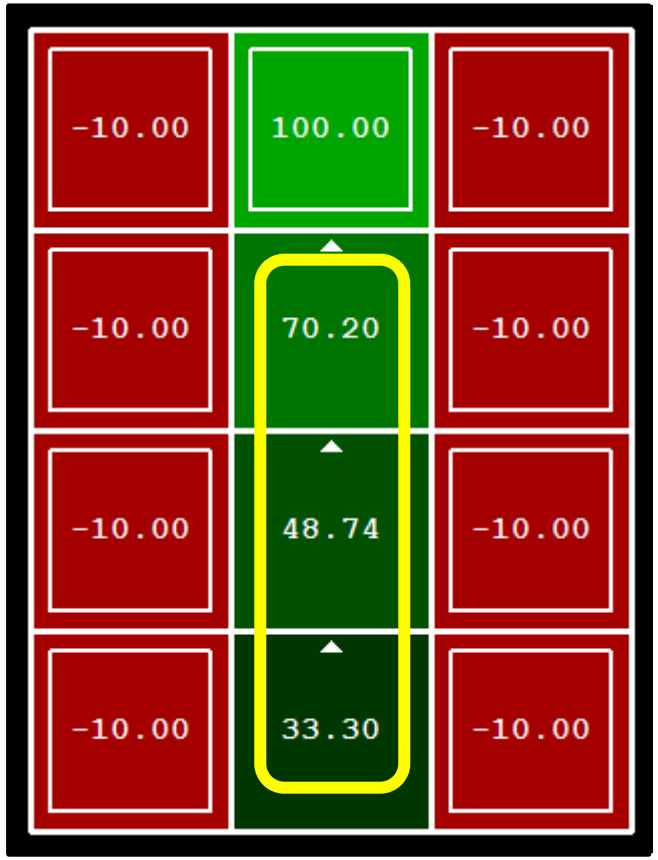
Always Go Right

$\gamma \cdot 100$

$\gamma^{2} \cdot 100$

$\gamma^{3} \cdot 100$

Always Go Forward



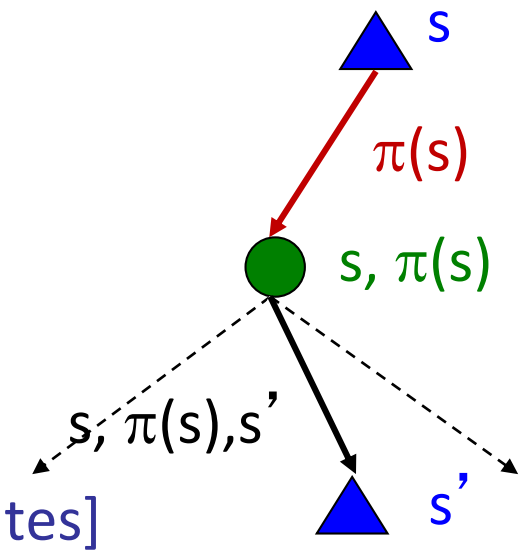
Policy Evaluation

- **Question:** How do we calculate the V^{π}'s for a fixed policy π?
- **Idea 1:** Turn recursive Bellman equations into updates (like ***value iteration***)

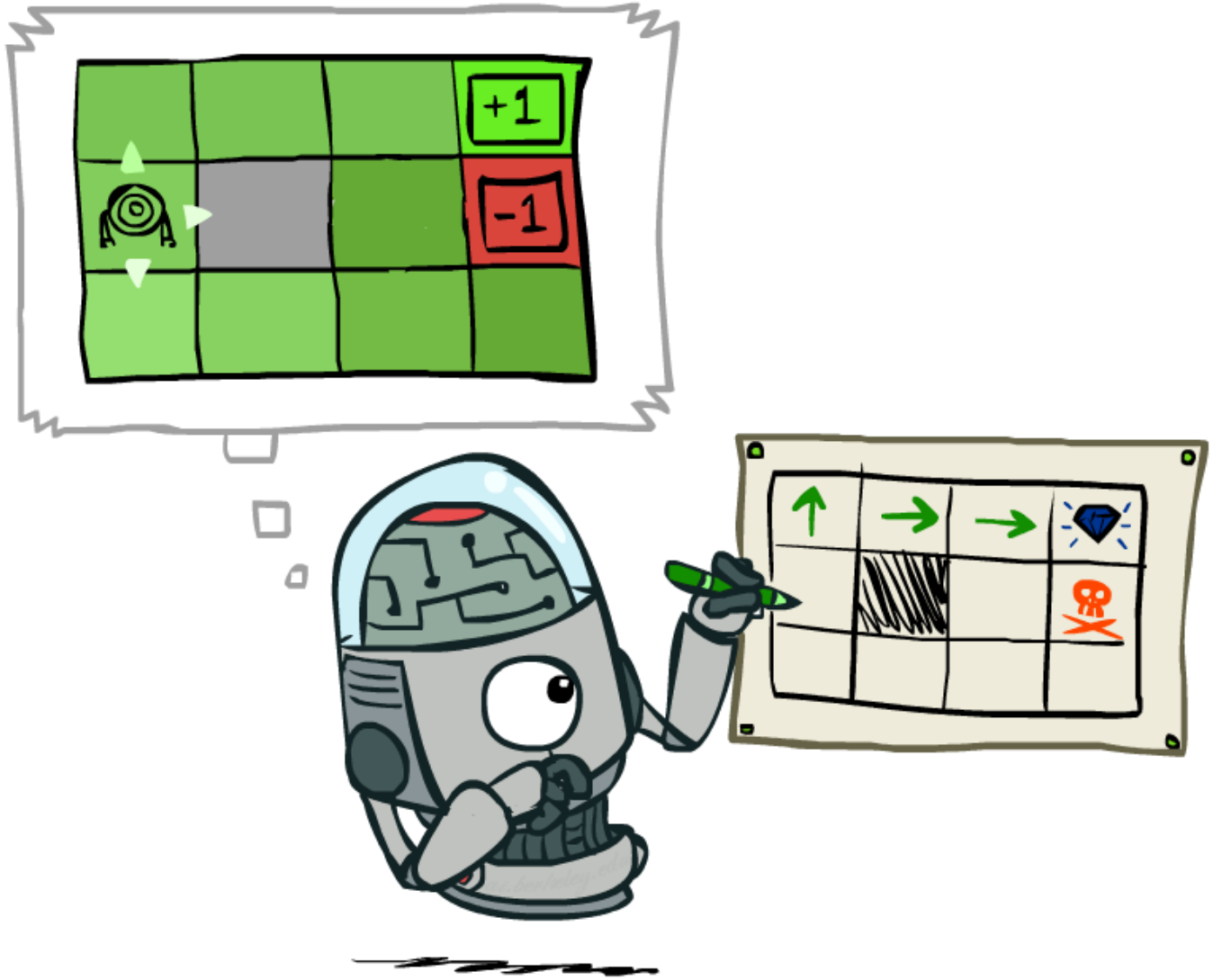
$$V_0^{\pi}(s) = 0$$

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

- Efficiency: $O(S^2)$ per iteration ($\forall$ **s** and **s'**) [We have **S** different states]

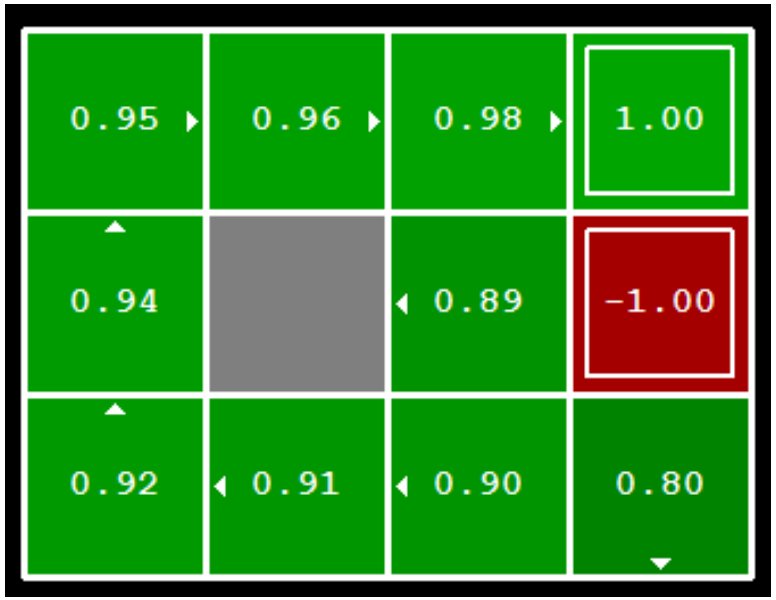


Policy Extraction



Computing Actions from Values

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



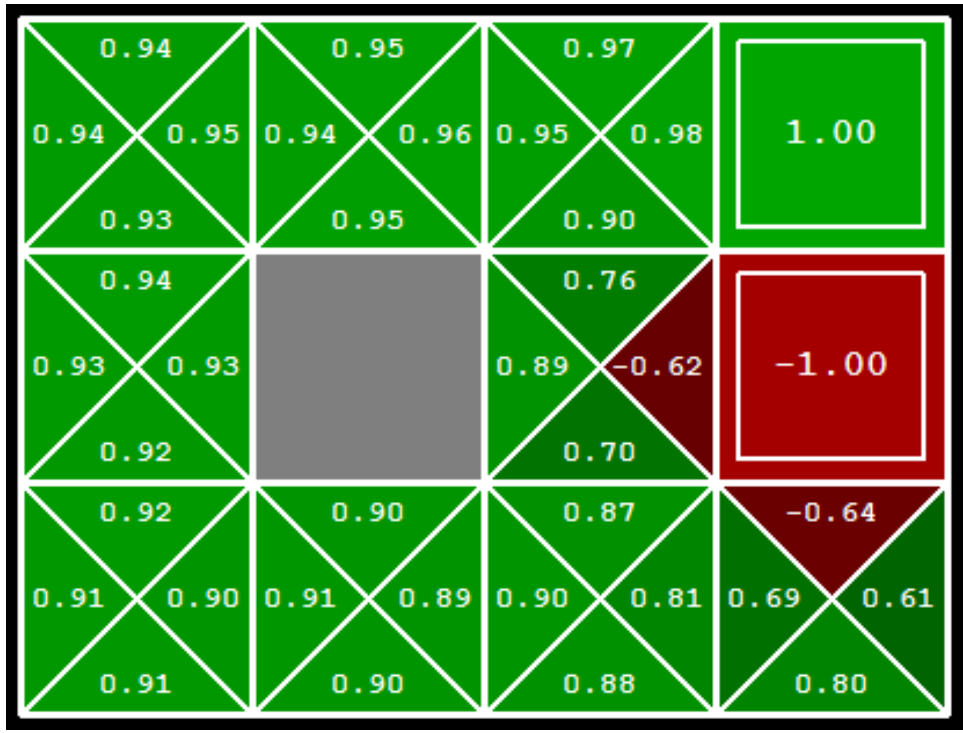
- **Question:** How should we act?
- **Answer:** It is not obvious!
- We need to do a mini-expectimax (one step):

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

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

Computing Actions from Q-Values

- Let us imagine we have the optimal q-values:

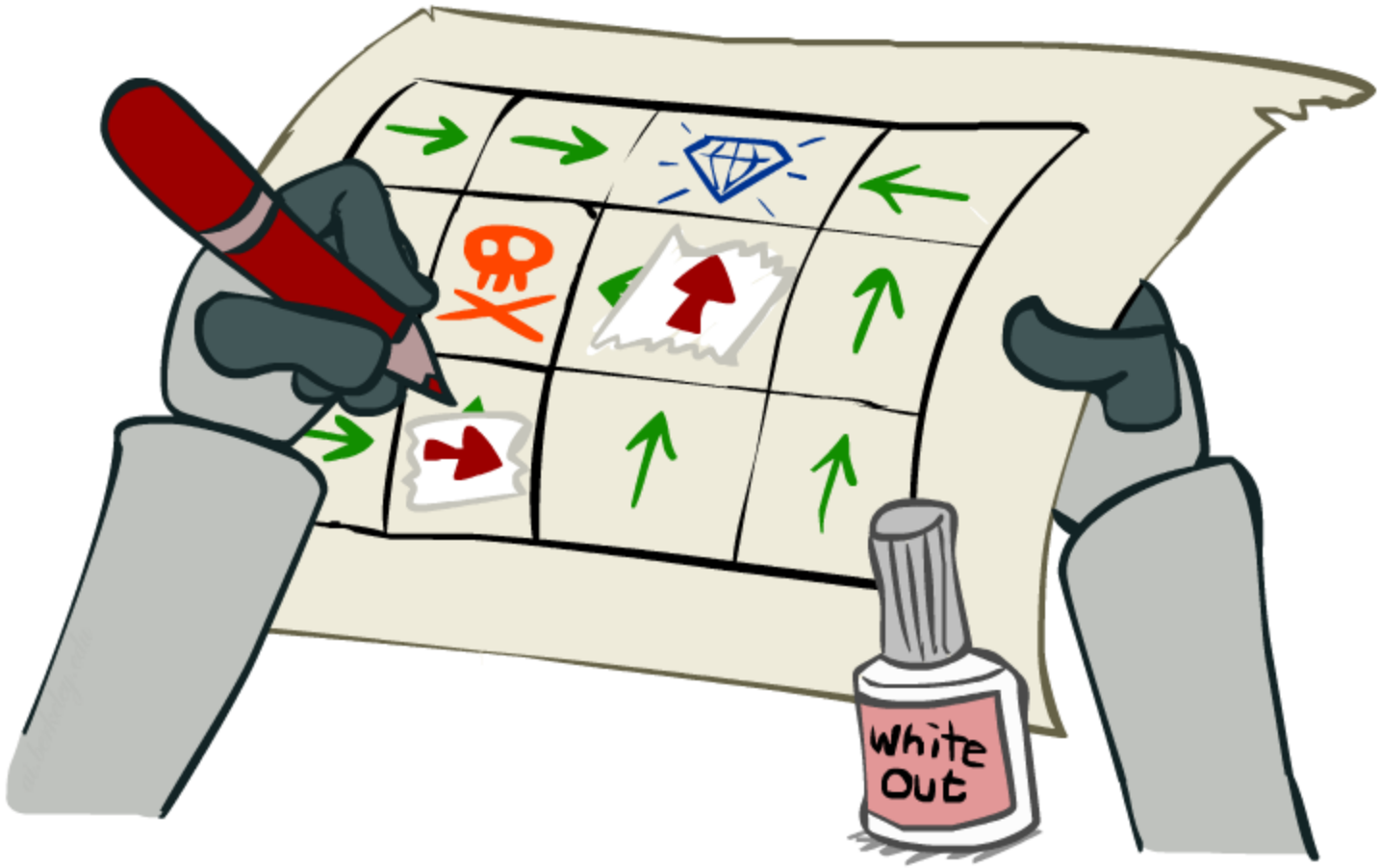


- **Question:** How should we act?
- **Answer:** Completely trivial to decide!

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

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

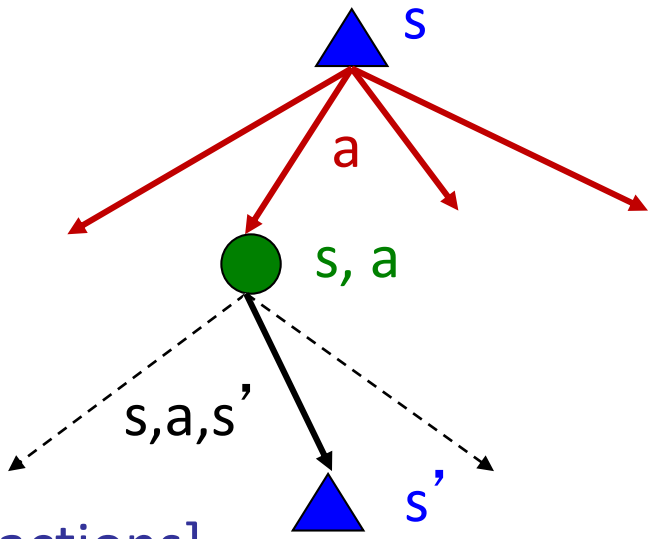
Policy Iteration



Problems with Value Iteration

- Value iteration repeats the ***Bellman updates***:

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



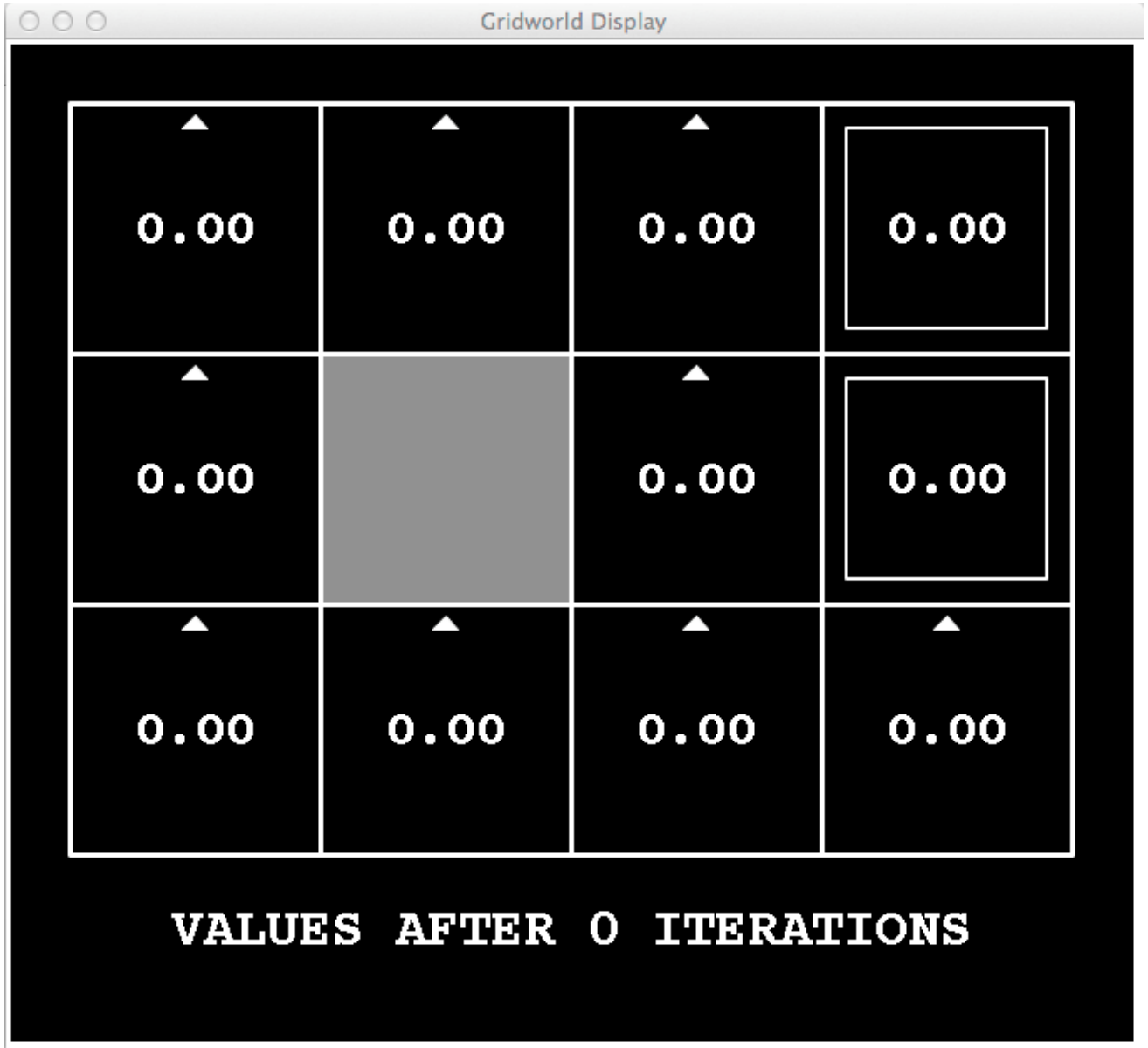
- **Problem 1:** It is slow – ***O(S²A)*** per iteration

(∀ **s**, **s'** and action **a**) [We have **S** different states and **A** different actions]

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

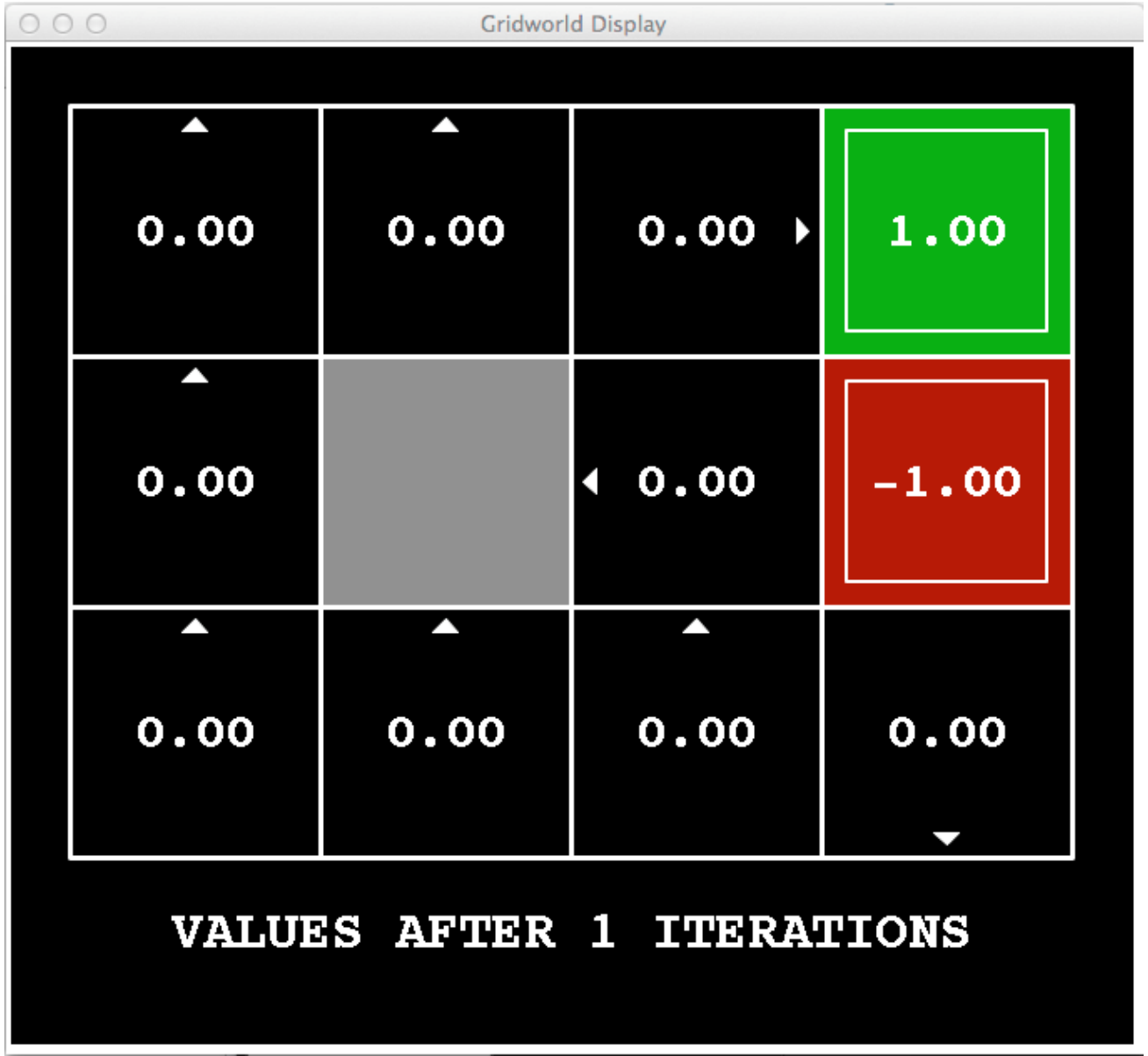
- **Problem 3:** The policy often converges long before the values

k = 0



Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 1



Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

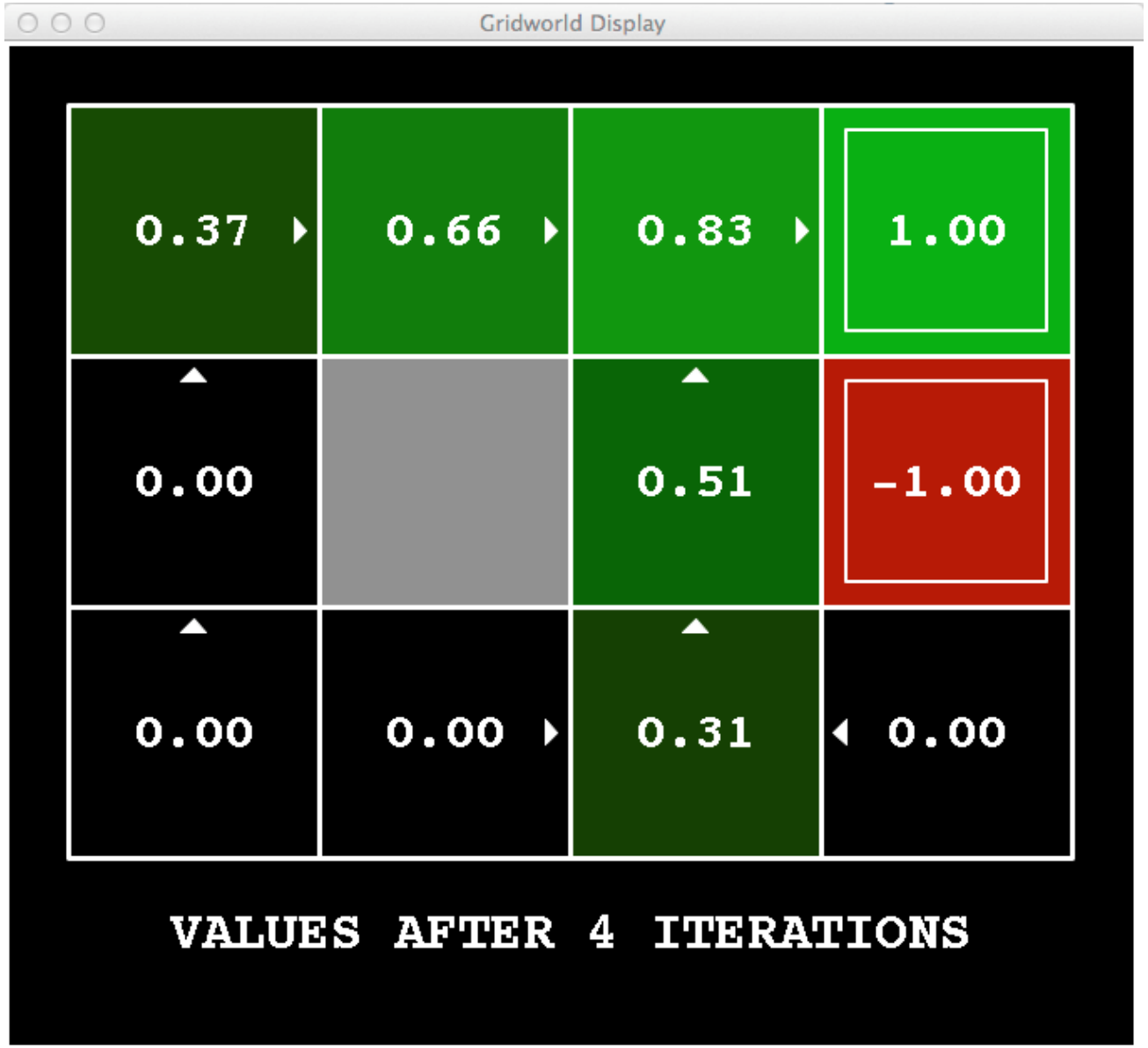
k = 2

Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 3

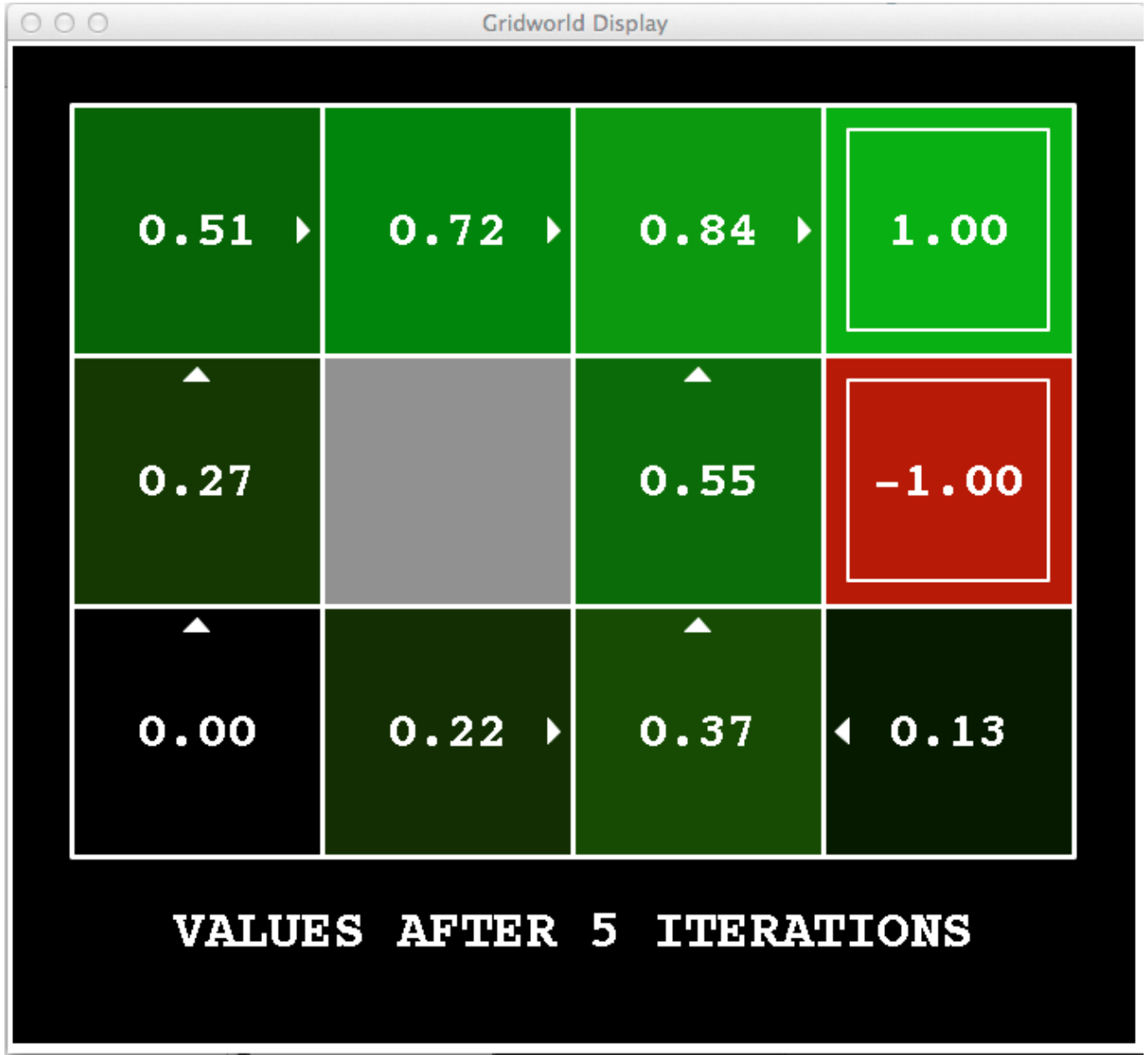
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 4



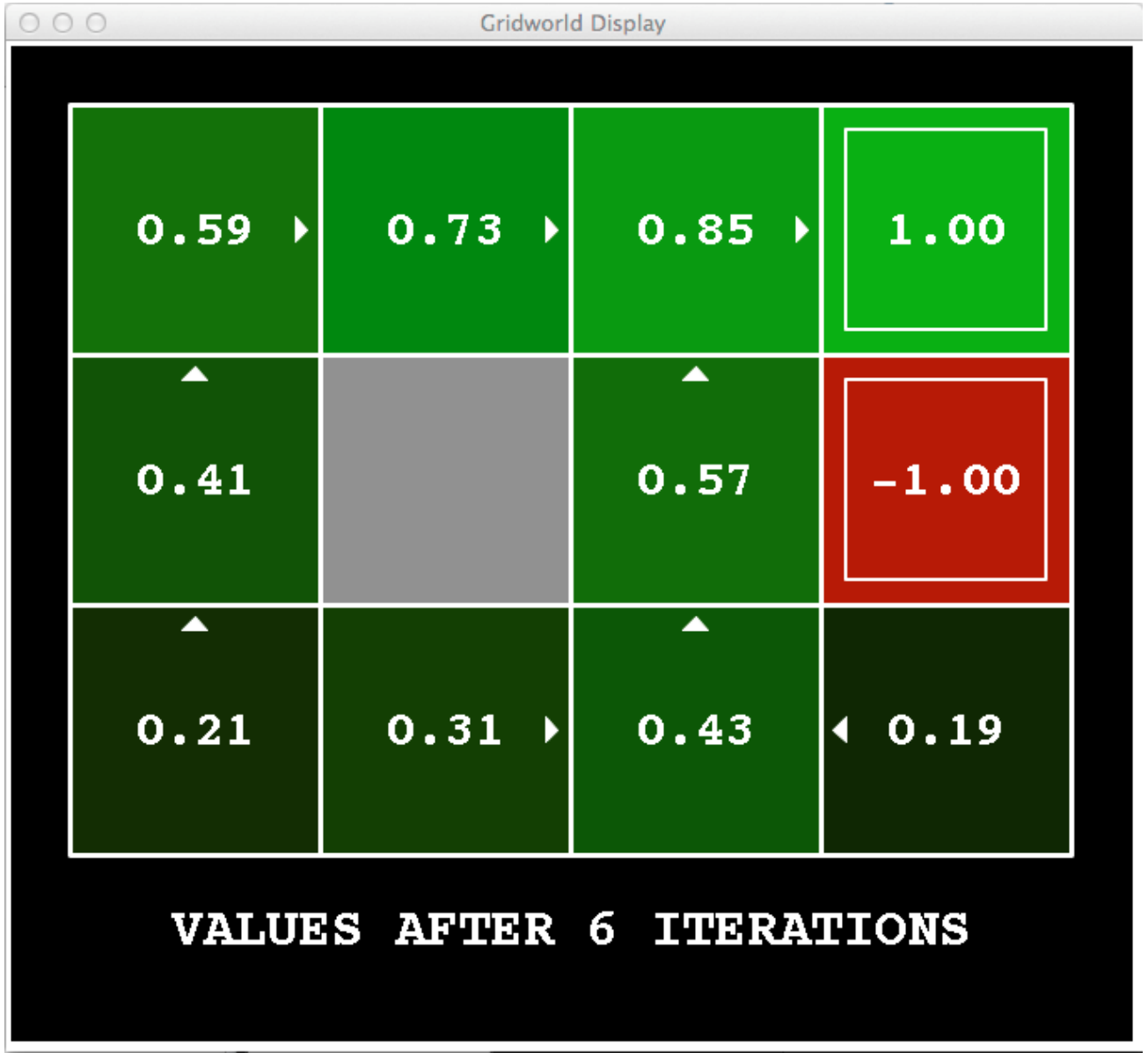
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 5



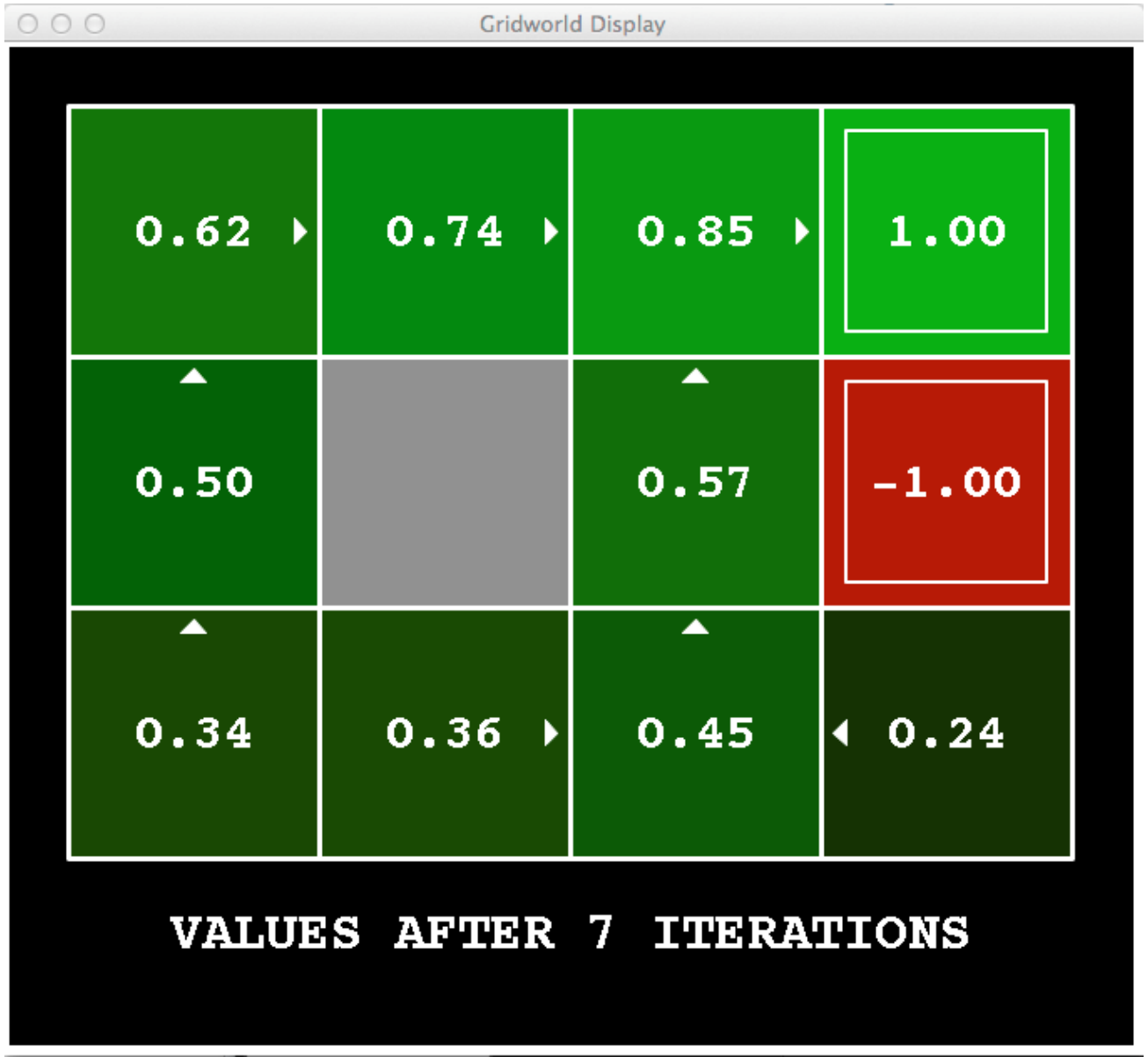
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 6



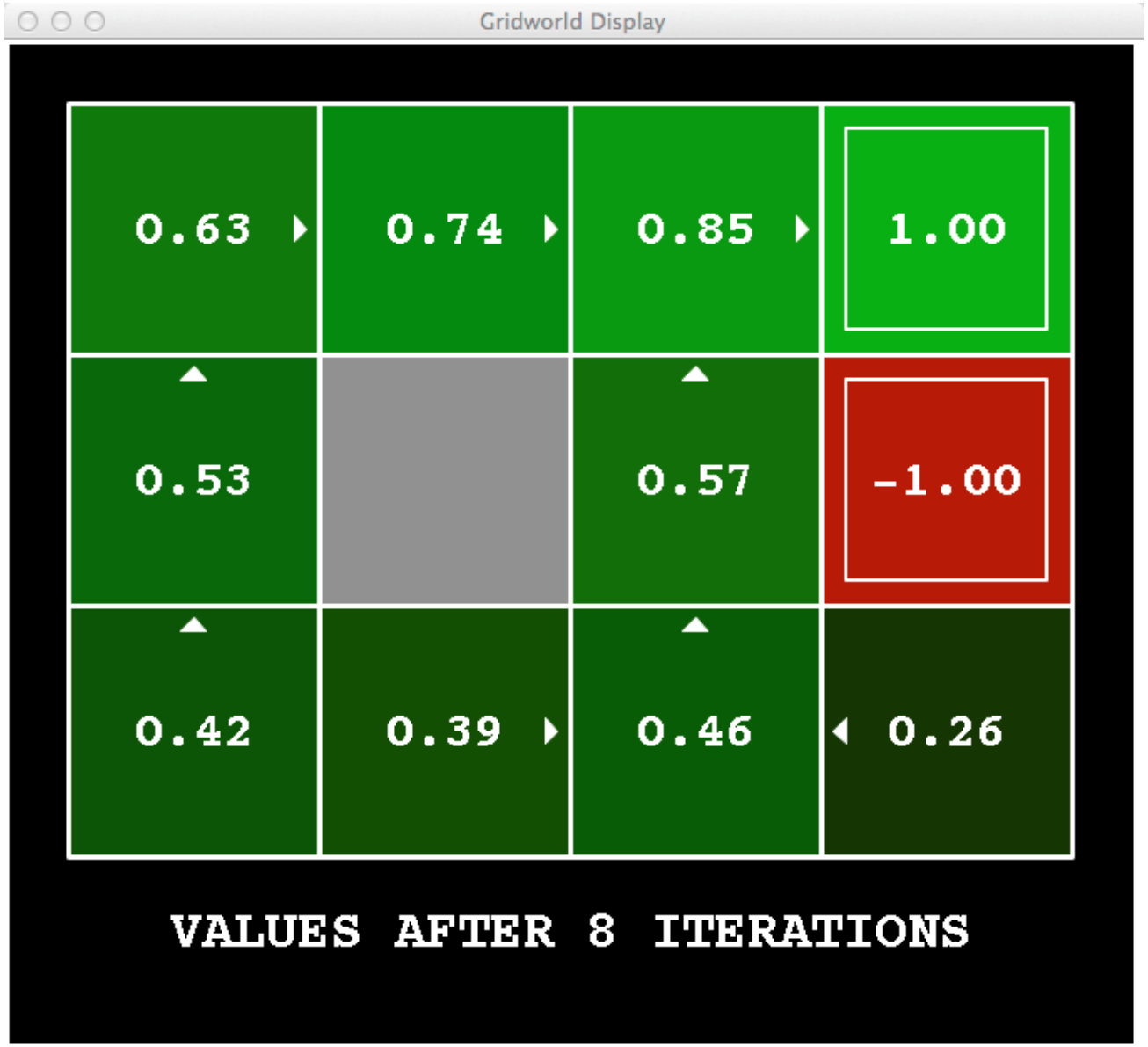
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 7



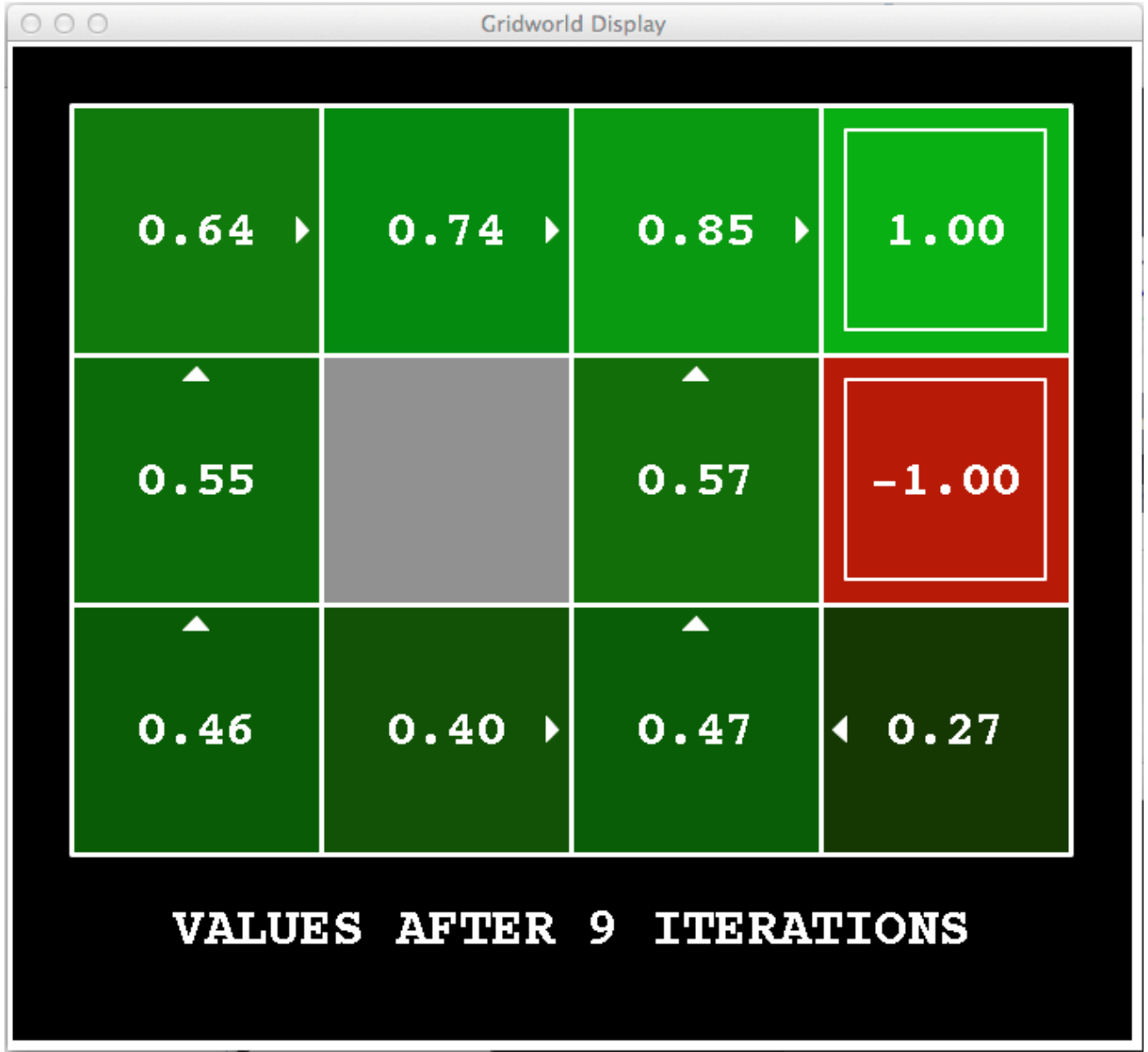
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 8



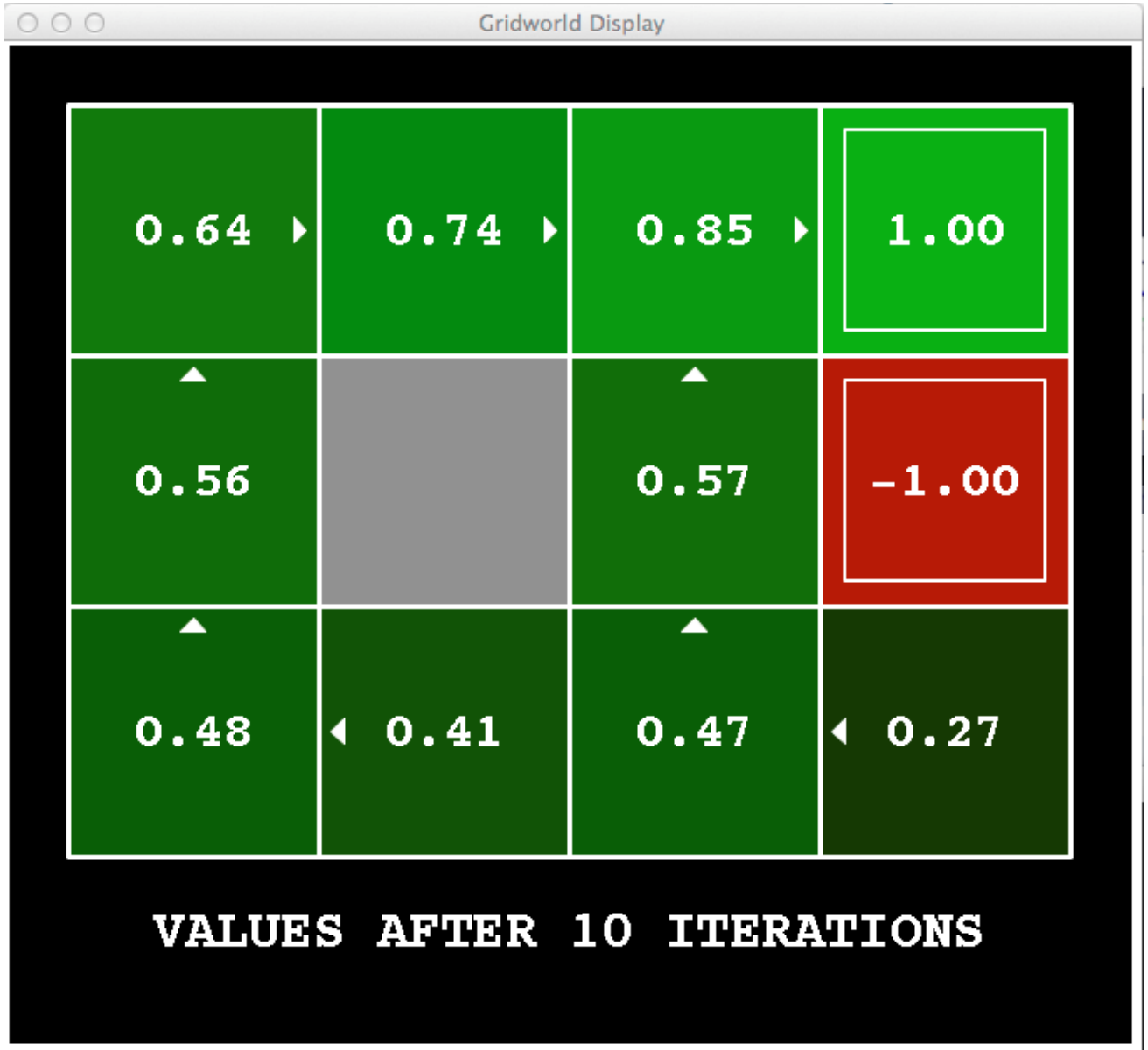
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 9



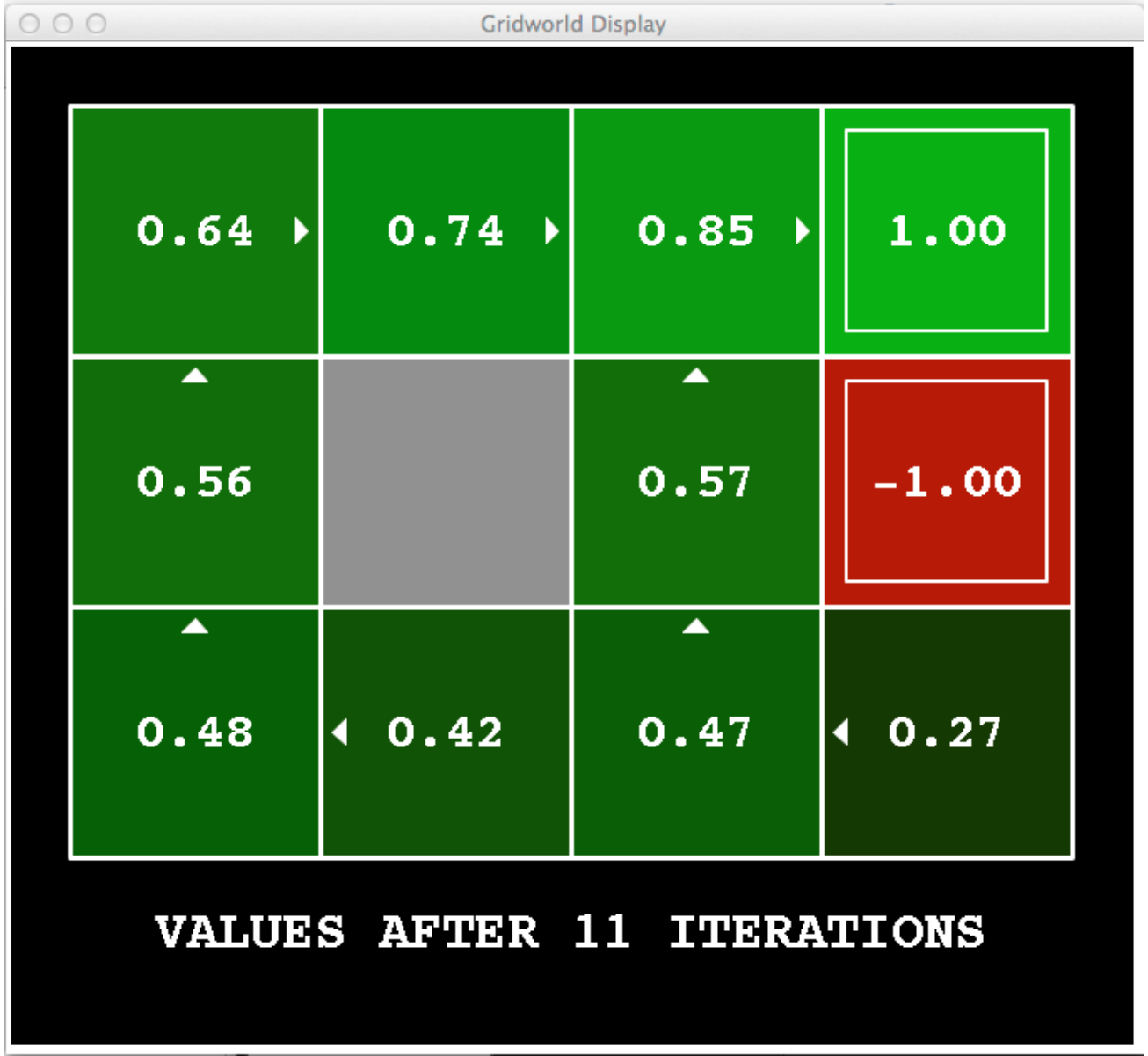
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 10



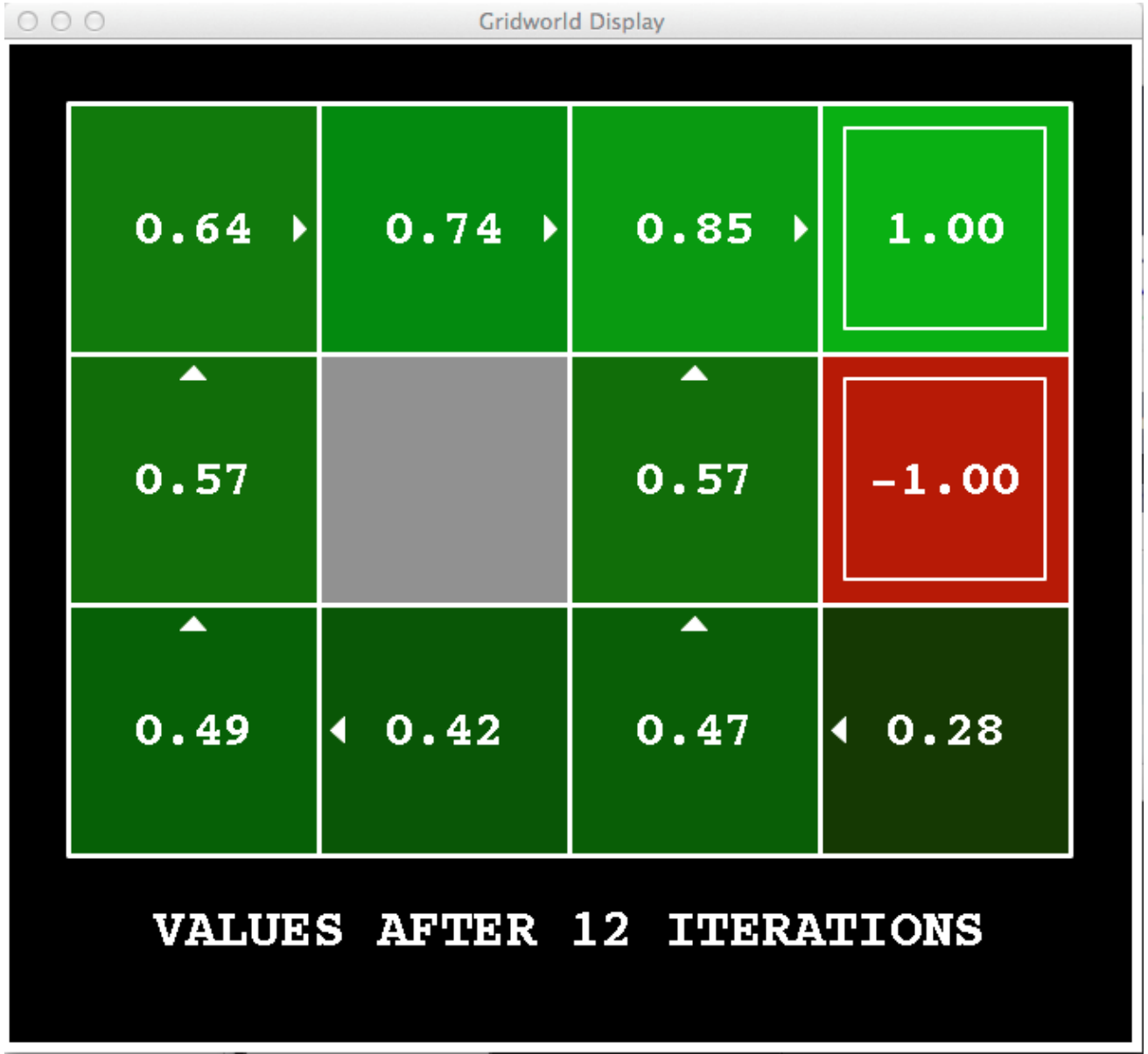
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 11



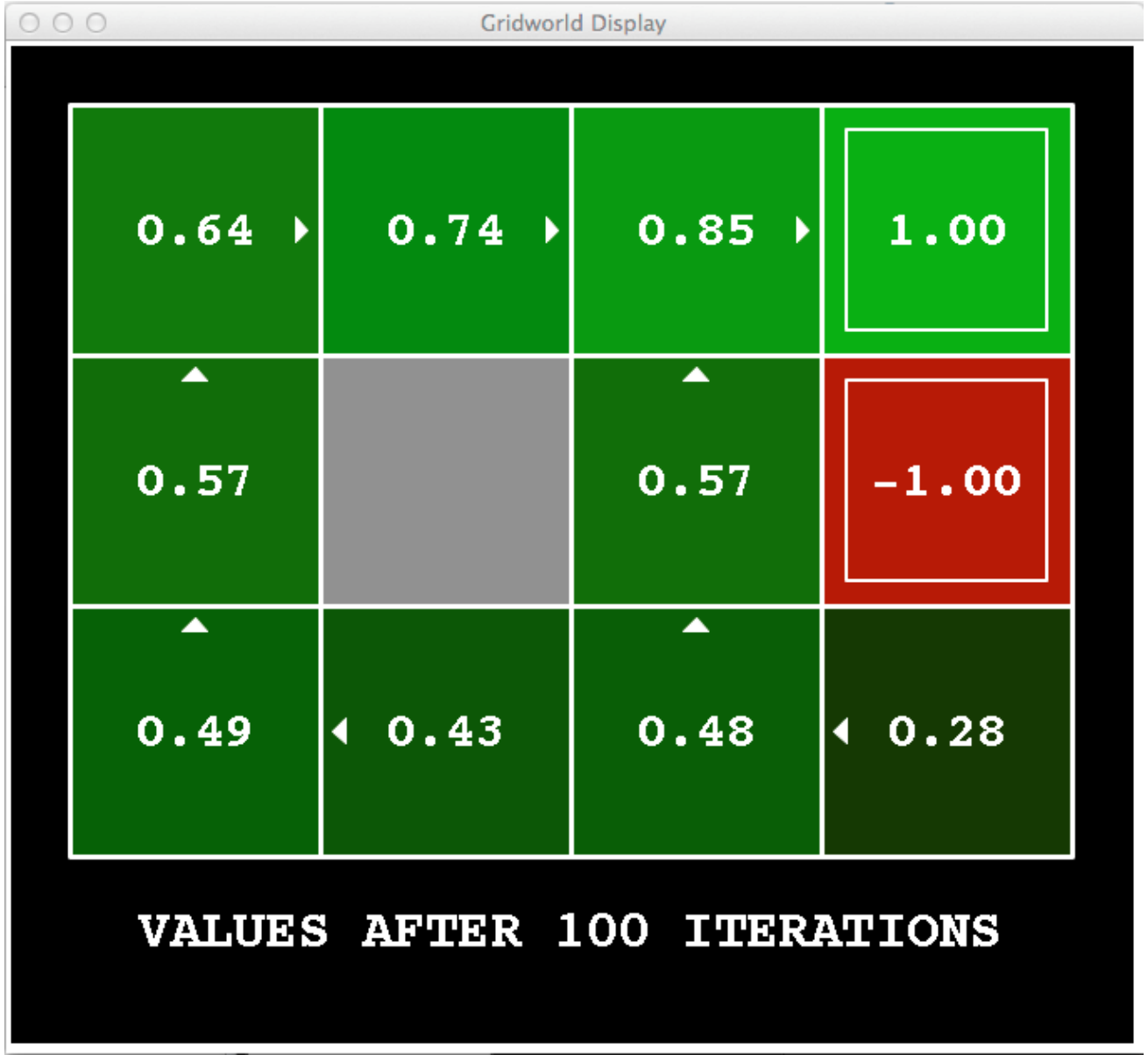
Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 12



Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

k = 100



Noise = 0.2
[$p(\uparrow) = 0.8, p(\leftarrow) = 0.1, p(\rightarrow) = 0.1$]
Discount ($\gamma = 0.9$)
Living reward ($r(s) = 0$)

Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: Calculate utilities for some fixed policy (not optimal utilities!) until convergence (or **solve the linear system of equations**!)
  - 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 is still optimal!
  - Can converge (much) faster under some conditions

Policy Iteration (Cont'd)

- 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:

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

- Remember that the number of possible policies for an MPD is: $(\#A)^{\#S}$ where #A and #S represents the number of actions and states, respectively.

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 do not 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 are done)
- Both are **dynamic programs** for solving MDPs

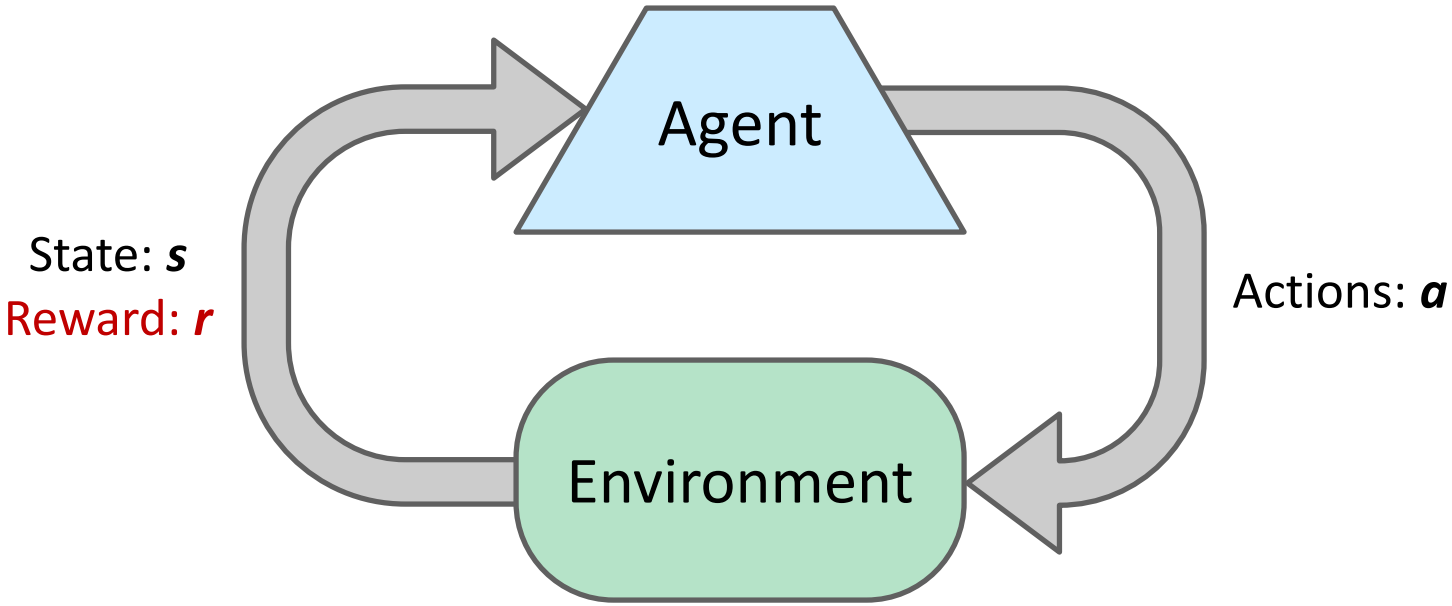
Summary: MDP Algorithms

- 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

Reinforcement Learning

Reinforcement Learning



- **Basic idea:**
  - Receive feedback in the form of rewards
  - Agent's utility is defined by the reward function
  - Must (learn to) act so as to maximize expected rewards
  - **All learning is based on observed samples** of outcomes!

Example: Learning to Walk

Initial

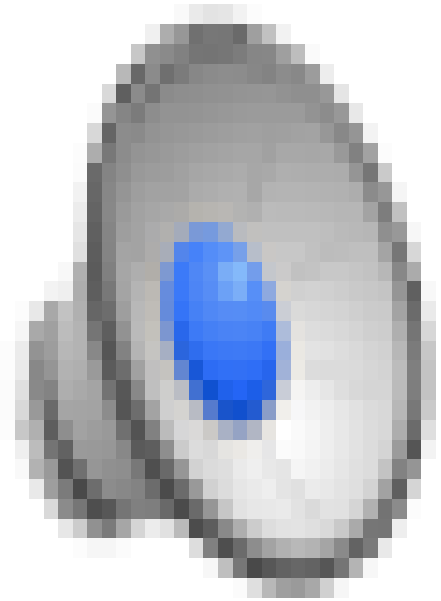
[Kohl and Stone, ICRA 2004]

Example: Learning to Walk (Cont'd)

Finished

[Kohl and Stone, ICRA 2004]

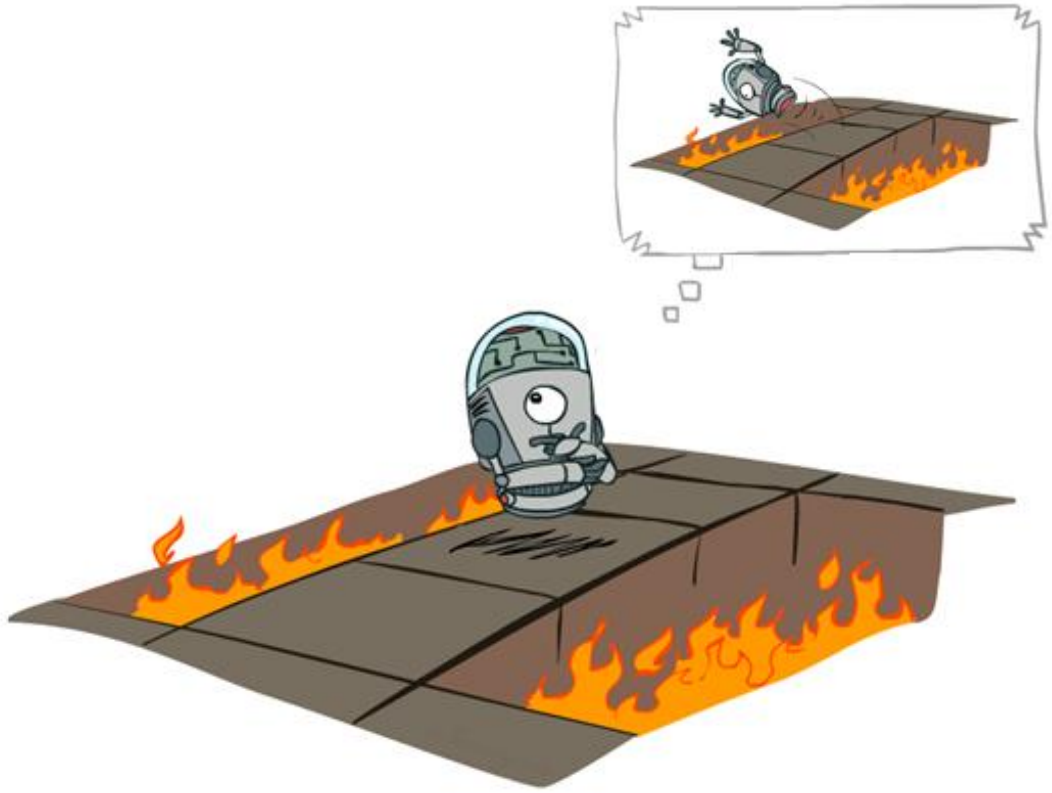
Video of Demo Crawler Bot



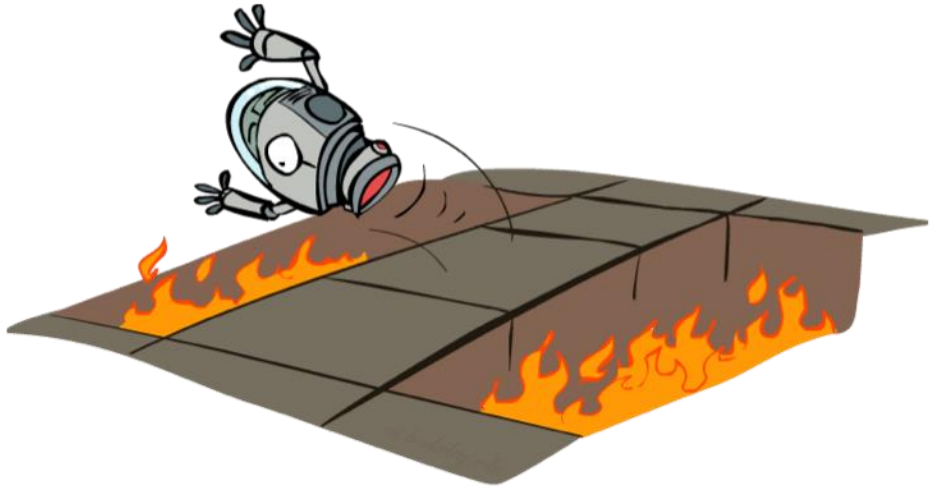
Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) A
  - A model $T(s, a, s')$
  - A reward function $R(s, a, s')$
- Still looking for a policy $\pi(s)$
- **New twist:** We do not know T or R
  - Therefore, we do not know which **states are good** or **what the actions do**
  - **Must actually try actions** and **states out to learn**

Offline (MDPs) vs. Online (RL)

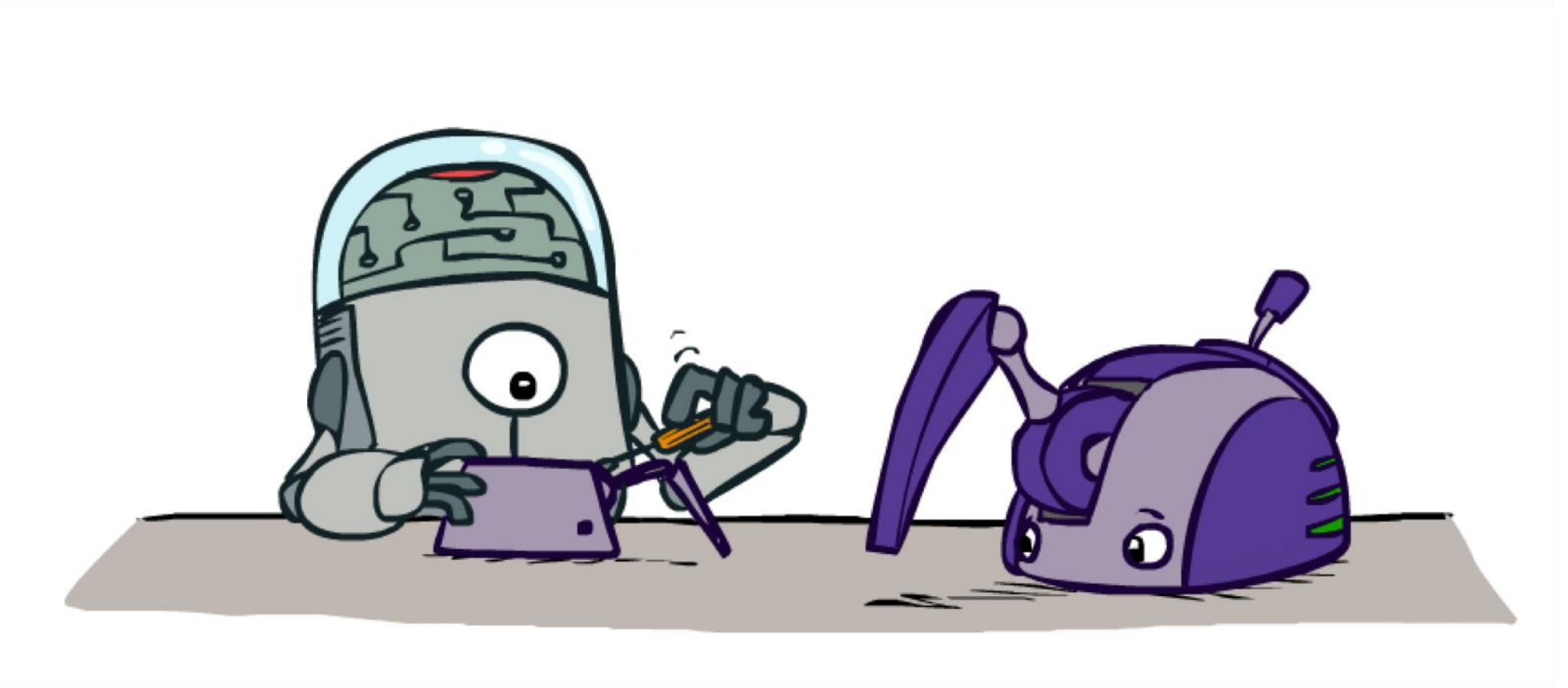


MDP: Offline Solution



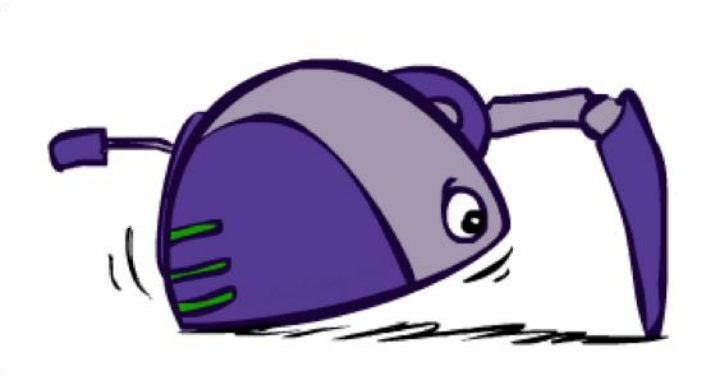
RL: Online Learning

Model-Based Learning



Model-Based Learning

- **Model-Based Idea:**
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

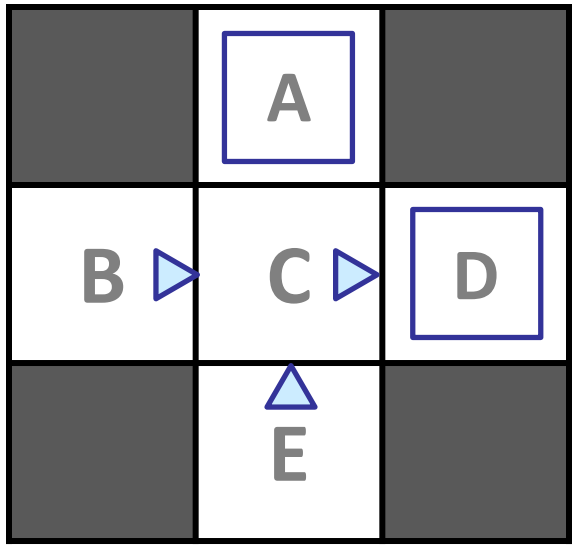


- **Step 1:** Learn empirical MDP model
  - Count outcomes ***s'*** for each ***s***, ***a***
  - Normalize to give an estimate of $\widehat{T}(s, a, s')$
  - Discover each $\widehat{R}(s, a, s')$ when we experience ***(s, a, s')***

- **Step 2:** Solve the **learned MDP**
  - For example, use value iteration, as before

Example: Model-Based Learning

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
…

$\hat{R}(s, a, s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Question: Are 4 episodes enough to "learn" our MDP?

Example: Expected Age

Goal: Compute expected age of the students in a class

Known P(A): Age distribution

$$E[A] = \sum_a P(a) \cdot a = 0.35 \times 20 + \dots$$

$E[A]$: Expected age

$P(a)$: Prob. of age

a: age

$\sum_a P(a) \cdot a$: Weighted average of age

Without ***P(A)***, instead collect samples **[a_1, a_2, … a_N]**

Unknown P(A): "Model Based"

$$\hat{P}(a) = \frac{\text{num}(a)}{N}$$

$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

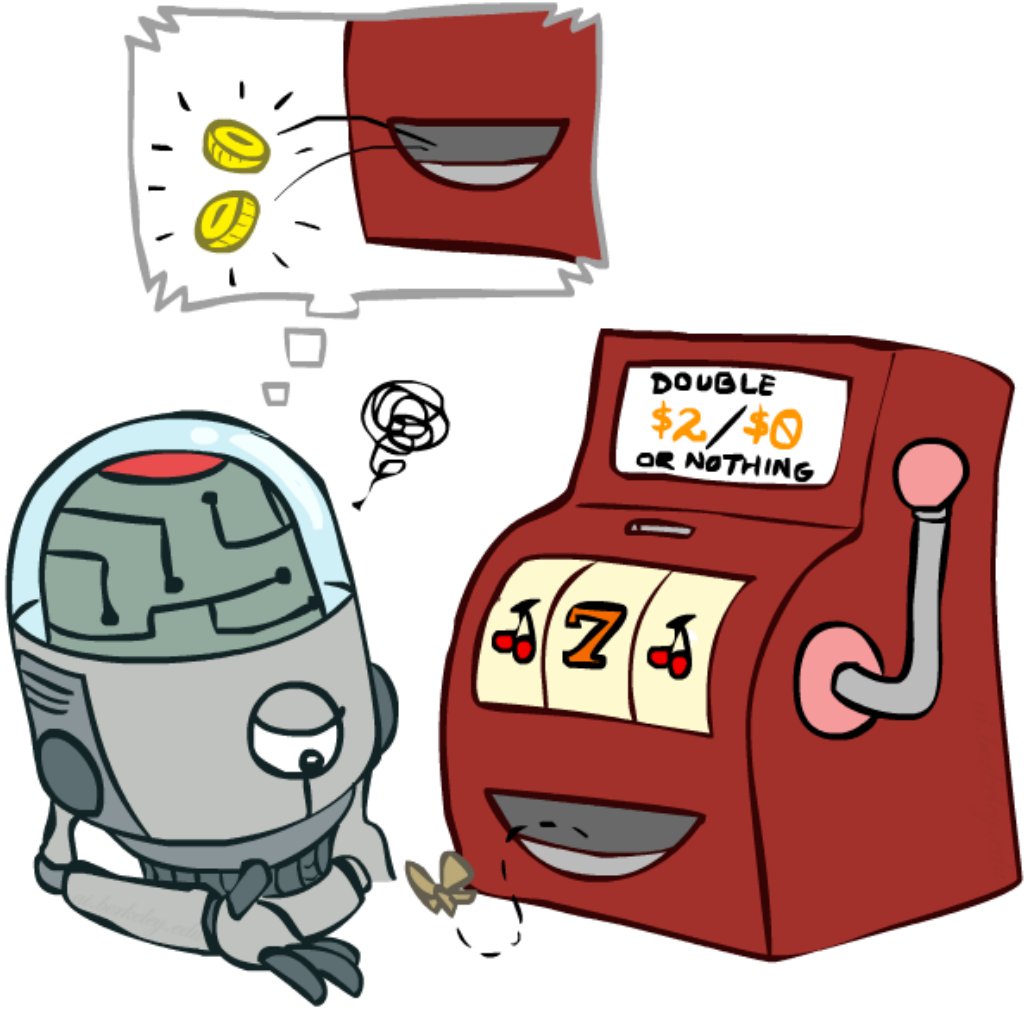
Why does this work? Because eventually you learn the **right model**.

Unknown P(A): "Model Free"

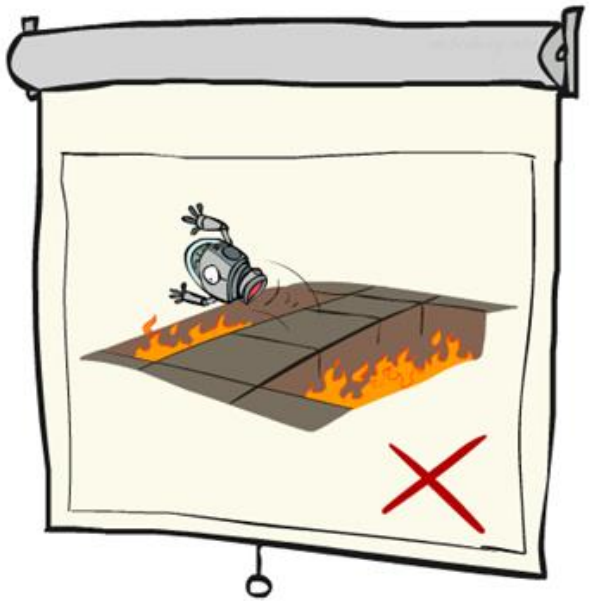
$$E[A] \approx \frac{1}{N} \sum_i a_i$$

Why does this work? Because **samples appear with the right frequencies**.

Model-Free Learning

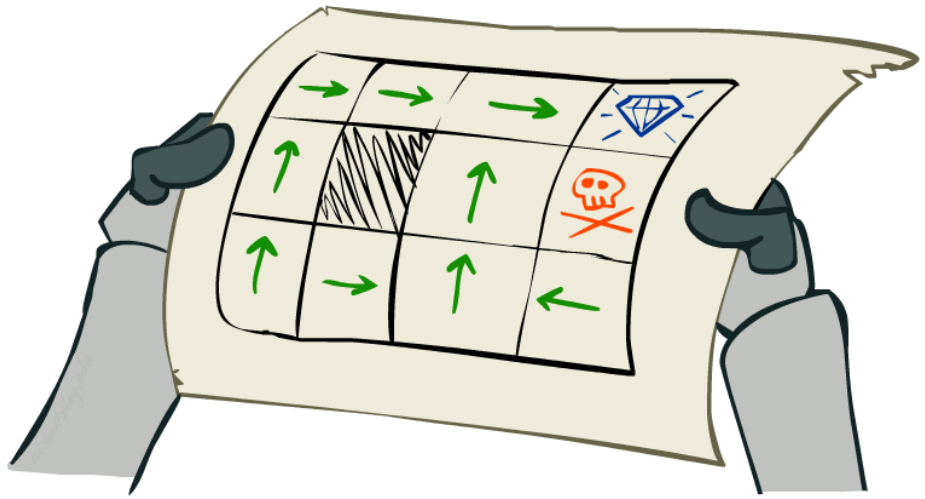


Passive Reinforcement Learning



Passive Reinforcement Learning (Cont'd)

- **Simplified task:** Policy evaluation
  - **Input:** A fixed policy $\pi(s)$
  - You do not know the transitions $T(s,a,s')$
  - You do not know the rewards $R(s,a,s')$
  - **Goal:** Learn the state values



- In this case:
  - Learner is "**along for the ride**"
  - **No choice about what actions to take** (cannot try actions!)
  - Just execute the policy and learn from experience
  - This is **NOT offline planning**! You actually take actions in the world.

Direct Evaluation

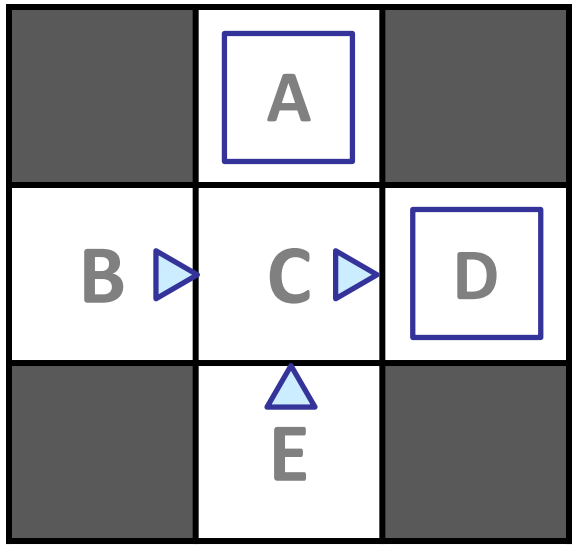
- **Goal:** Compute values for each state under π

- **Idea:** Average together observed sample values
  - Act according to π
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - **Average those samples**

- This is called **direct evaluation**

Example: Direct Evaluation

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

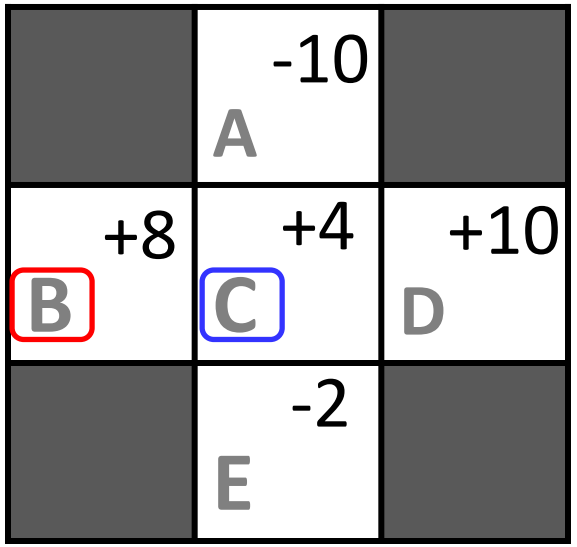
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



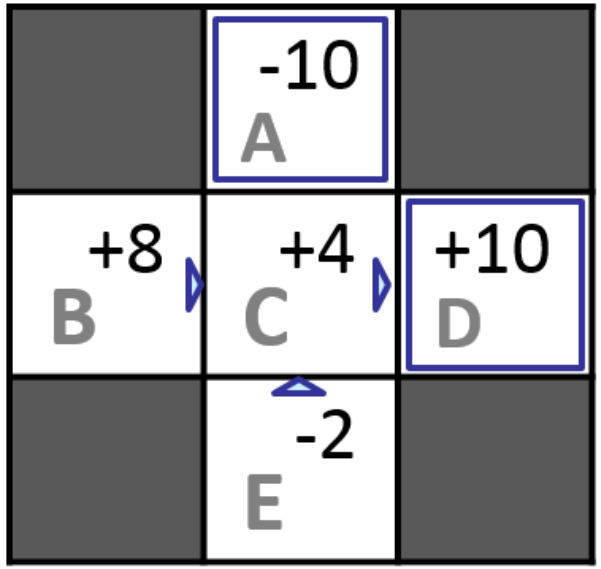
$V^{\pi}(B) = ((+8) + (+8)) / 2 = 8$

$V^{\pi}(C) = ((+9) + (+9) + (+9) + (-11)) / 4 = 4$

Problems with Direct Evaluation

- **Question:** What is good about direct evaluation?
  - It is easy to understand
  - It does not require any knowledge of ***T*** and ***R***
  - It eventually computes the correct average values, using just sample transitions
- **Question:** What bad about it?
  - It wastes information about state connections
  - Each state must be learned separately
  - So, it takes a long time to learn

Output Values



Question: *If **B** and **E** both go to **C** under this policy, how can their values be different?*

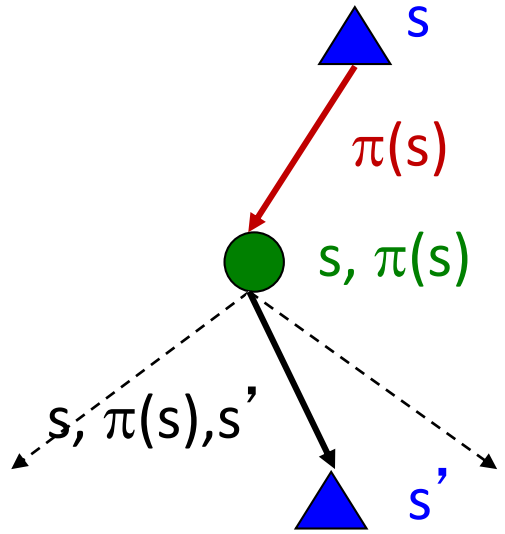
Answer: *The value of state **E** is not taking advantage of the value in state **C** which contradicts Bellman equations!*

Why Not Use Policy Evaluation?

- Simplified Bellman updates calculate ***V*** for a fixed policy:
  - Each round, replace ***V*** with a one-step-look-ahead layer over ***V***:

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

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} [\quad\quad + \gamma V_k^{\pi}(s')]$$



  - This approach fully exploited the connections between the states
  - Unfortunately, we need ***T*** and ***R*** to do it!

- **Key question:** How can we do this update to ***V*** without knowing ***T*** and ***R***?
  - In other words, how do we take a weighted average without knowing the weights?

Sample-Based Policy Evaluation?

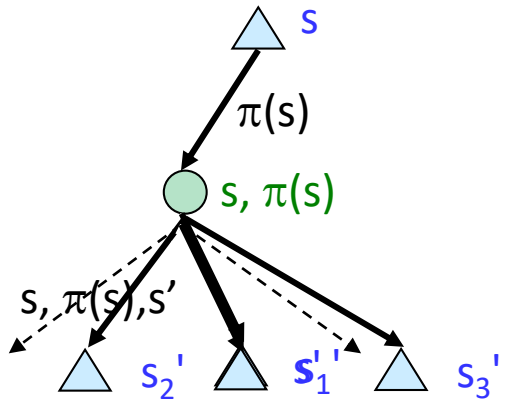
- We want to improve our estimate of ***V*** by computing these averages:

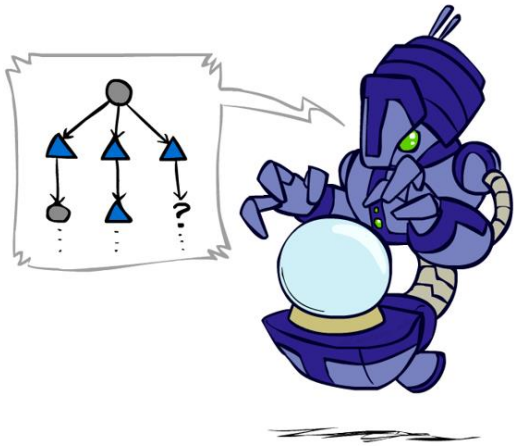
$$V^{\pi}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{k}(s')]$$

- **Idea:** Take samples of outcomes ***s'*** (by doing the action ***a***!) and average

$$sample_1 = R(s, \pi(s), s'_1) + \gamma V^{\pi}_{k}(s'_1)$$
$$sample_2 = R(s, \pi(s), s'_2) + \gamma V^{\pi}_{k}(s'_2)$$
$$\cdots$$
$$sample_n = R(s, \pi(s), s'_n) + \gamma V^{\pi}_{k}(s'_n)$$

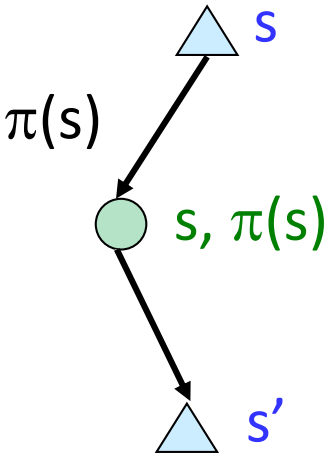
$$V^{\pi}_{k+1}(s) \leftarrow \frac{1}{n} \sum_{i} sample_i$$





TD Learning (Cont'd)

- **Big idea:** ***Learn from every experience***!
  - Update ***V(s)*** each time we experience a transition ***(s, a, s', r)***
  - Likely outcomes ***s'*** will contribute updates more often



- ***Temporal difference learning*** of values
  - **Policy still fixed (π)**, still doing evaluation!
  - Move values toward value of whatever successor occurs: ***Running average***

Inaccurate at the beginning of the update process

Sample of ***V(s)***: $$sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

Update to ***V(s)***: $$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$$

Reordering

Same update: $$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

No need to store experience (Episodes).
We store only values V(s).

Exponential Moving Average

- ***Exponential moving average***
  - The running interpolation update: New average $\bar{x}_n = (1-\alpha)\cdot \bar{x}_{n-1} + \alpha \cdot x_n$ (Current average, New sample)
  - Makes ***recent*** samples ***more important***:

Newer samples get higher weights: More accurate

Older samples get lower weights: Less accurate

$$\bar{x}_n = \frac{x_n + (1-\alpha)\cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + \dots}$$

$(1-\alpha)^L$

Normalizing factor

$$\bar{x}_{n-1} = (1-\alpha)\cdot \bar{x}_{n-2} + \alpha \cdot x_{n-1}$$

$$\bar{x}_n = (1-\alpha)\cdot \left[(1-\alpha)\cdot \bar{x}_{n-2} + \alpha \cdot x_{n-1}\right] + \alpha \cdot x_n$$

$$\bar{x}_n = (1-\alpha)\cdot \left[(1-\alpha)\cdot \left[(1-\alpha)\cdot \bar{x}_{n-3} + \alpha \cdot x_{n-2}\right] + \alpha \cdot x_{n-1}\right] + \alpha \cdot x_n$$

  - **Forgets about the past** (***distant past values were wrong anyway***)

- Decreasing ***learning rate*** (α) can give converging averages

$$\sum_{k=}^{\infty} \alpha_k = \infty$$

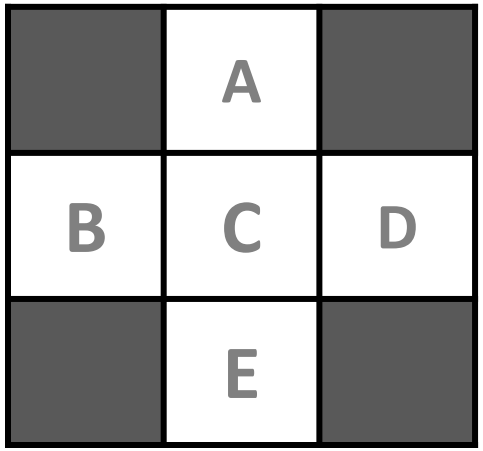
$$\sum_{k=}^{\infty} \alpha_k^2 < \infty$$

Question: How to choose α?

Answer: Many choices: 1) $\alpha(n) = 1/n$, 2) $\alpha(n_s) = 1/n_s$ where n_s is the number of times state ***s*** was observed.

Example: TD Learning

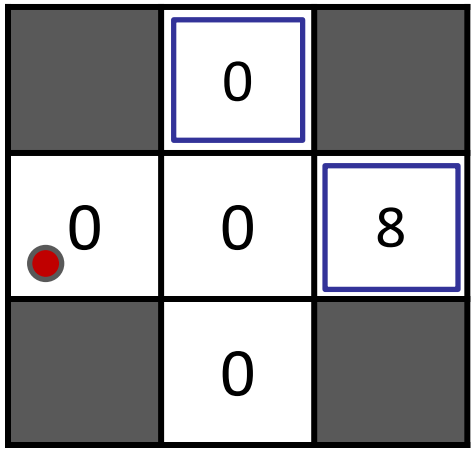
States



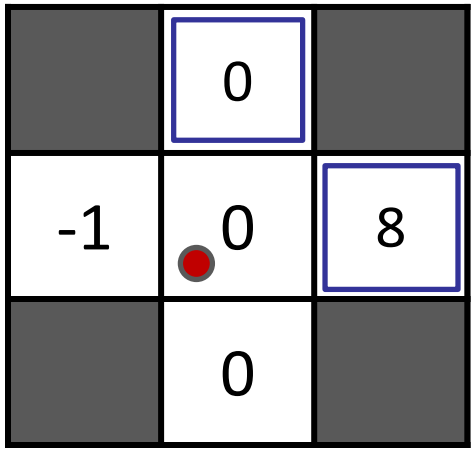
Assume: $\gamma = 1$, $\alpha = 1/2$

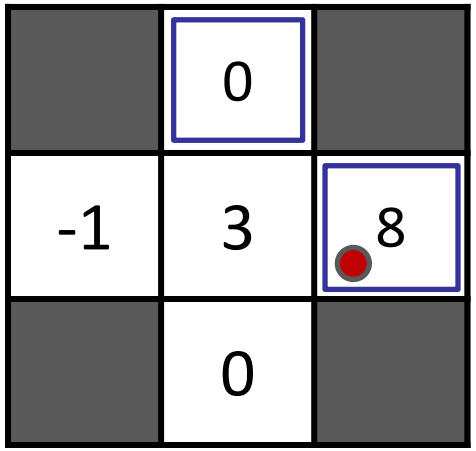
Observed Transitions

B, east, C, -2



C, east, D, -2





$V^{\pi}(B) \leftarrow (1 - 0.5) \cdot 0 + 0.5 \cdot [-2 + 1 \cdot 0] = -1$

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

$V^{\pi}(C) \leftarrow (1 - 0.5) \cdot 0 + 0.5 \cdot [-2 + 1 \cdot 8] = 3$

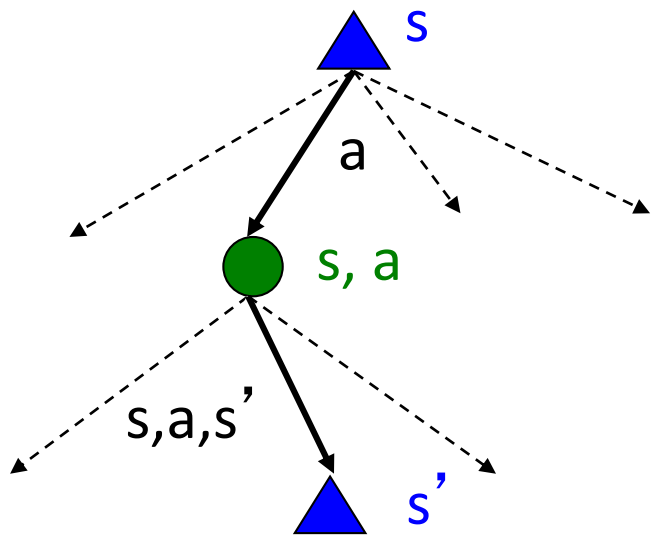
Problems with TD Value Learning

- TD value leaning is a ***model-free*** way to do policy evaluation, mimicking Bellman updates with ***running sample averages***

- However, if we want to ***turn values into a (new) policy***, we ***will not be able to do so***:

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

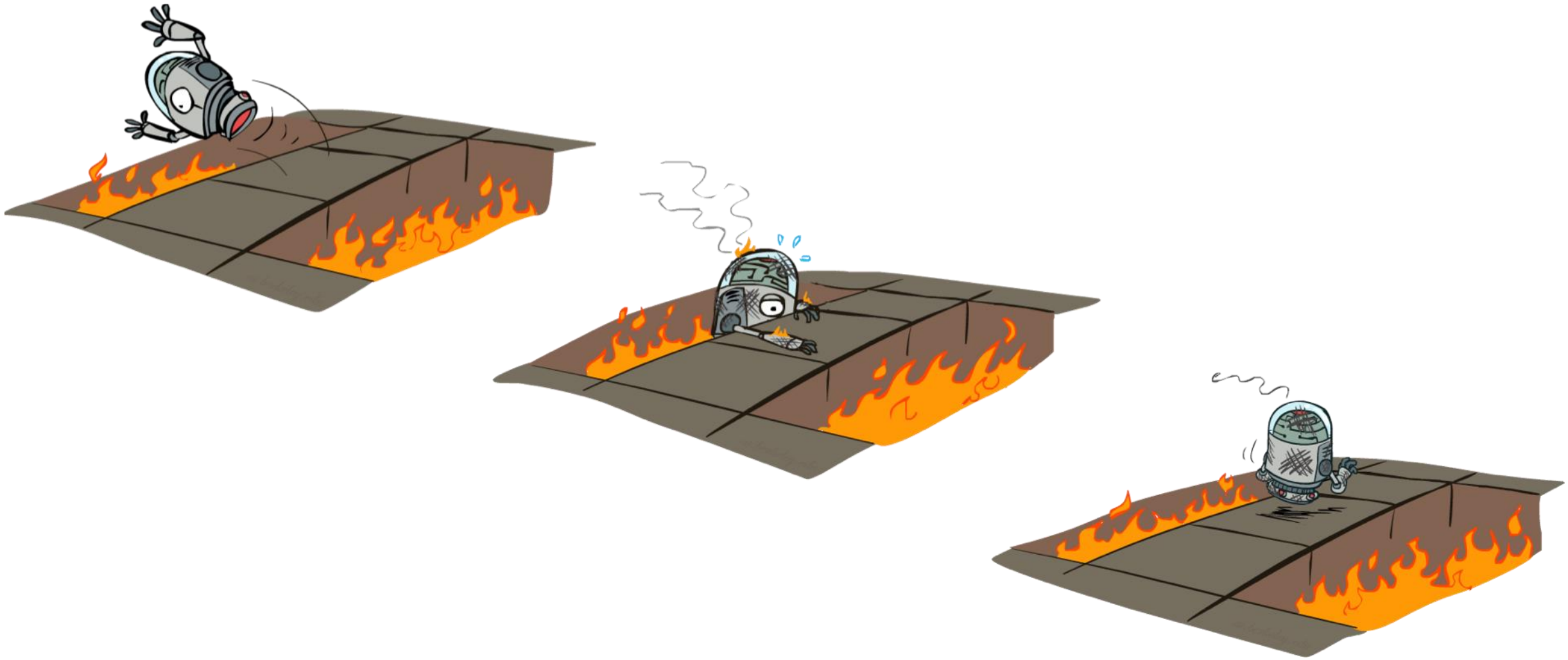
$$Q(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right]$$



- **Idea:** Learn Q-values, not values

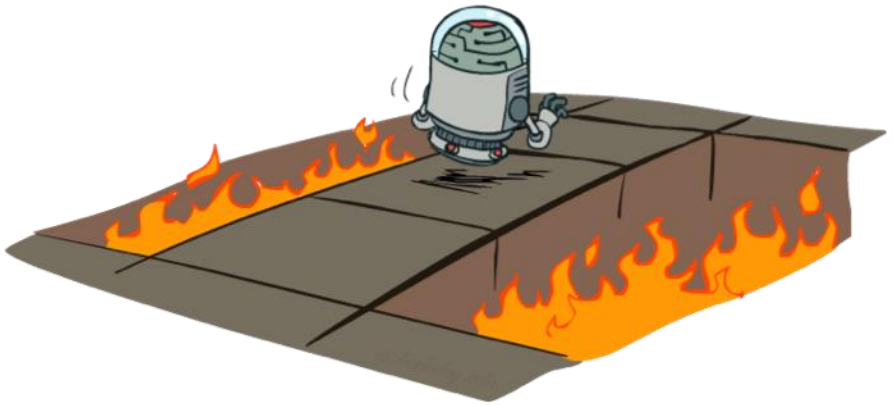
- Makes ***action selection model-free*** too!

Active Reinforcement Learning



Active Reinforcement Learning (Cont'd)

- Full reinforcement learning: Optimal policies (like value iteration)
  - You do not know the transitions ***T(s, a, s')***
  - You do not know the rewards ***R(s, a, s')***
  - You ***choose*** the ***actions now***
  - **Goal:** Learn the ***optimal policy / values***



- In this case:
  - Learner ***makes choices***!
  - Fundamental tradeoff: ***Exploration*** vs. ***exploitation***
  - This is ***NOT offline planning***! You actually take actions in the world and find out what happens…

Detour: Q-Value Iteration

- **Value iteration:** Find successive (***depth-limited***) values
  - Start with $V_0(s) = 0$, which we know is right
  - Given V_k, calculate the depth ***k+1*** values for all states:

The ***max*** operator **cannot be approximated by a running average!**

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

- But ***Q-values are more useful***, so compute them instead
  - Start with $Q_0(s, a) = 0$, which we know is right
  - Given Q_k, calculate the depth ***k+1*** q-values for all ***q-states***:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

$V_k(s')$ — One quantity

An average quantity can be approximated by a running average!

Q-Learning Algorithm

- **Q-Learning Algorithm:** ***Sample-based*** Q-value iteration

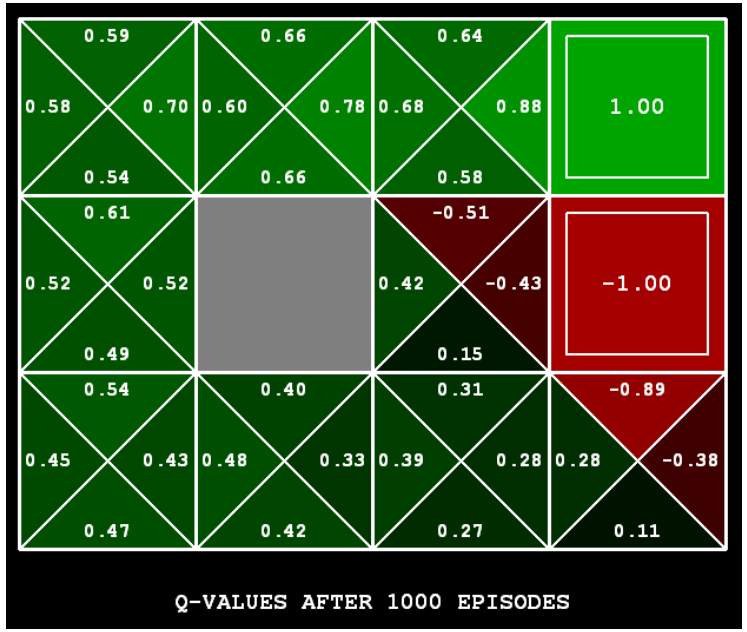
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

- Learn ***Q(s, a)*** values as you go
  - Receive a sample ***(s, a, s', r)***
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

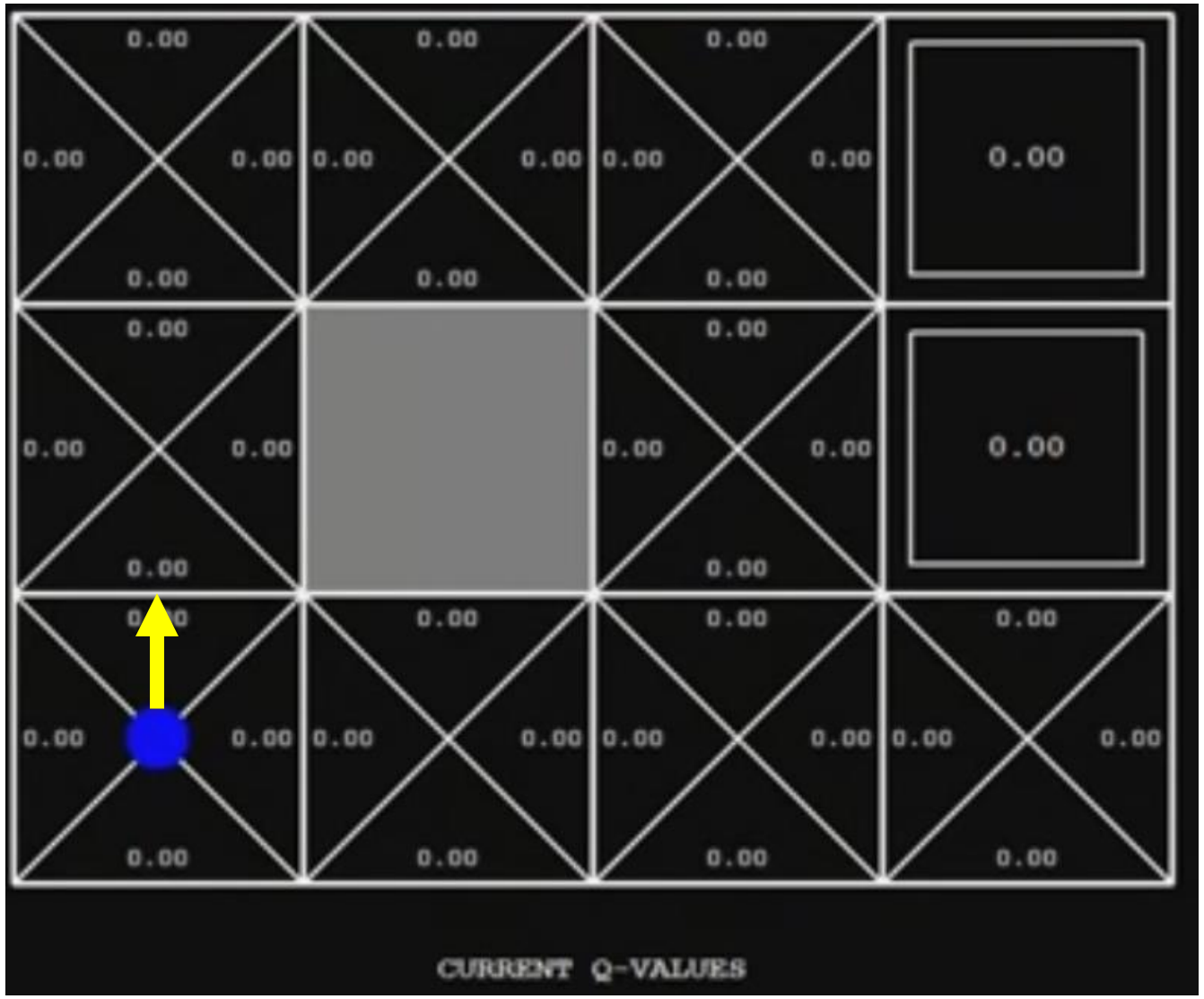
$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

  - Incorporate the new estimate into a running average:

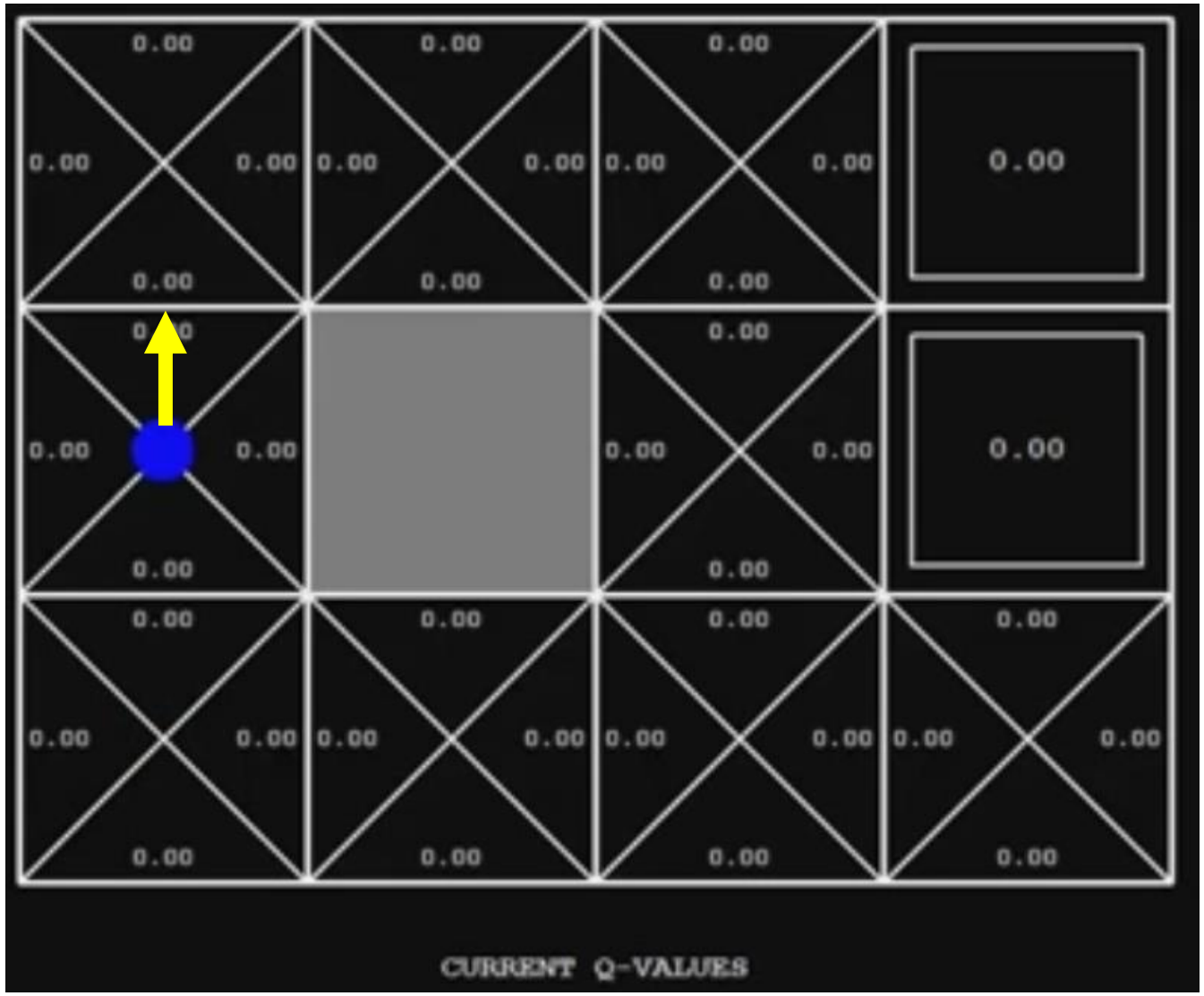
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$



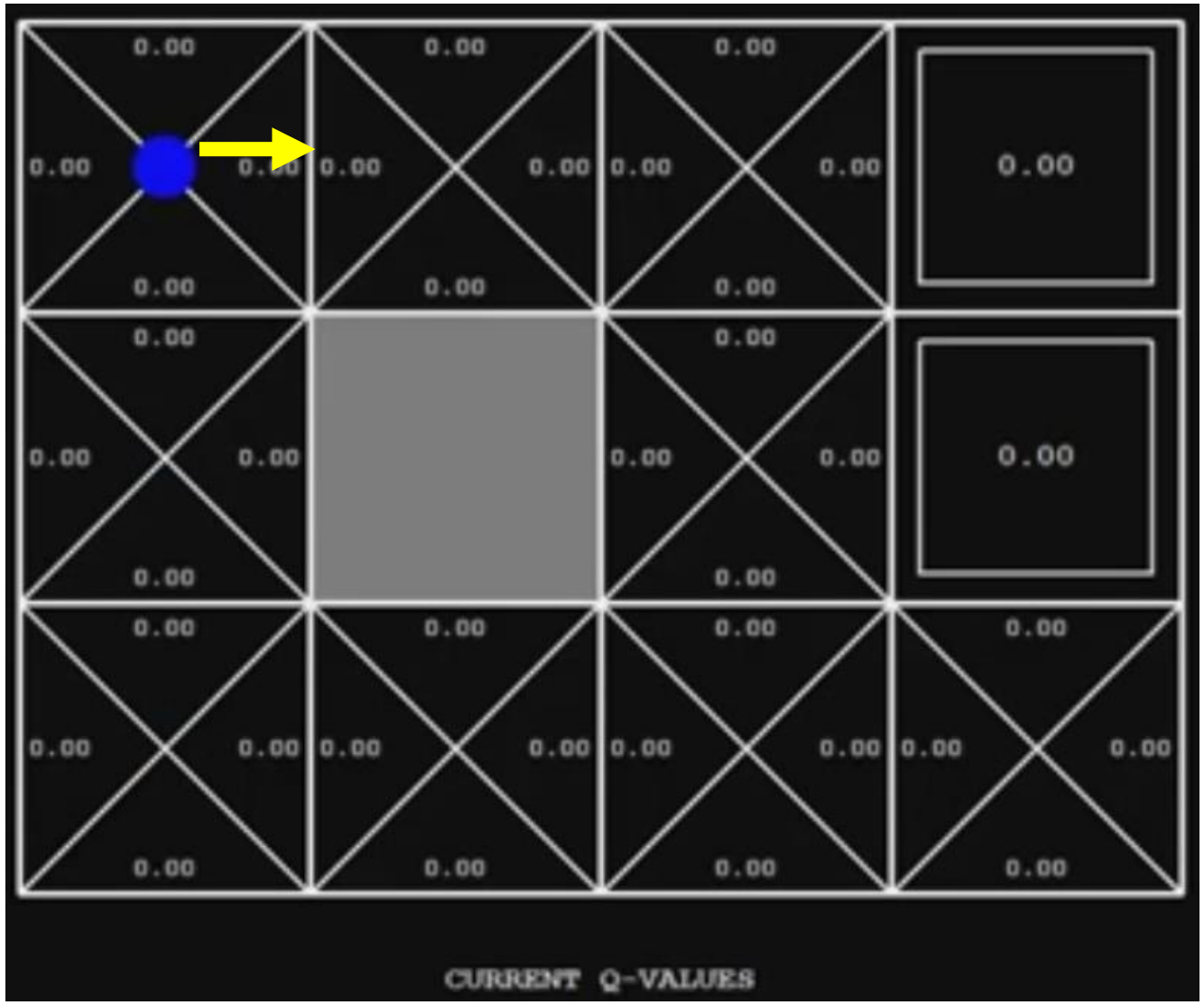
Q-Learning Algorithm (Cont'd)



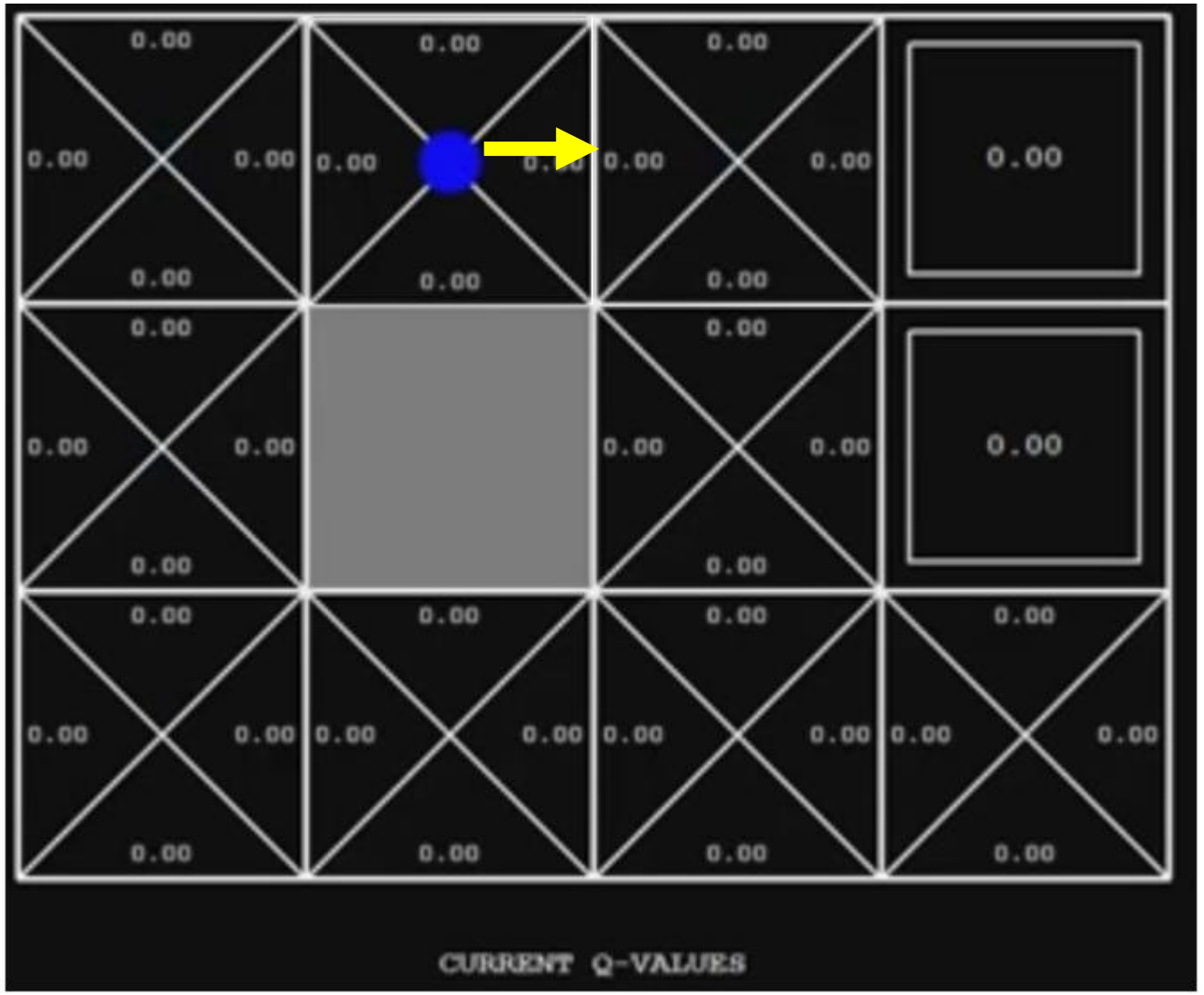
Q-Learning Algorithm (Cont'd)



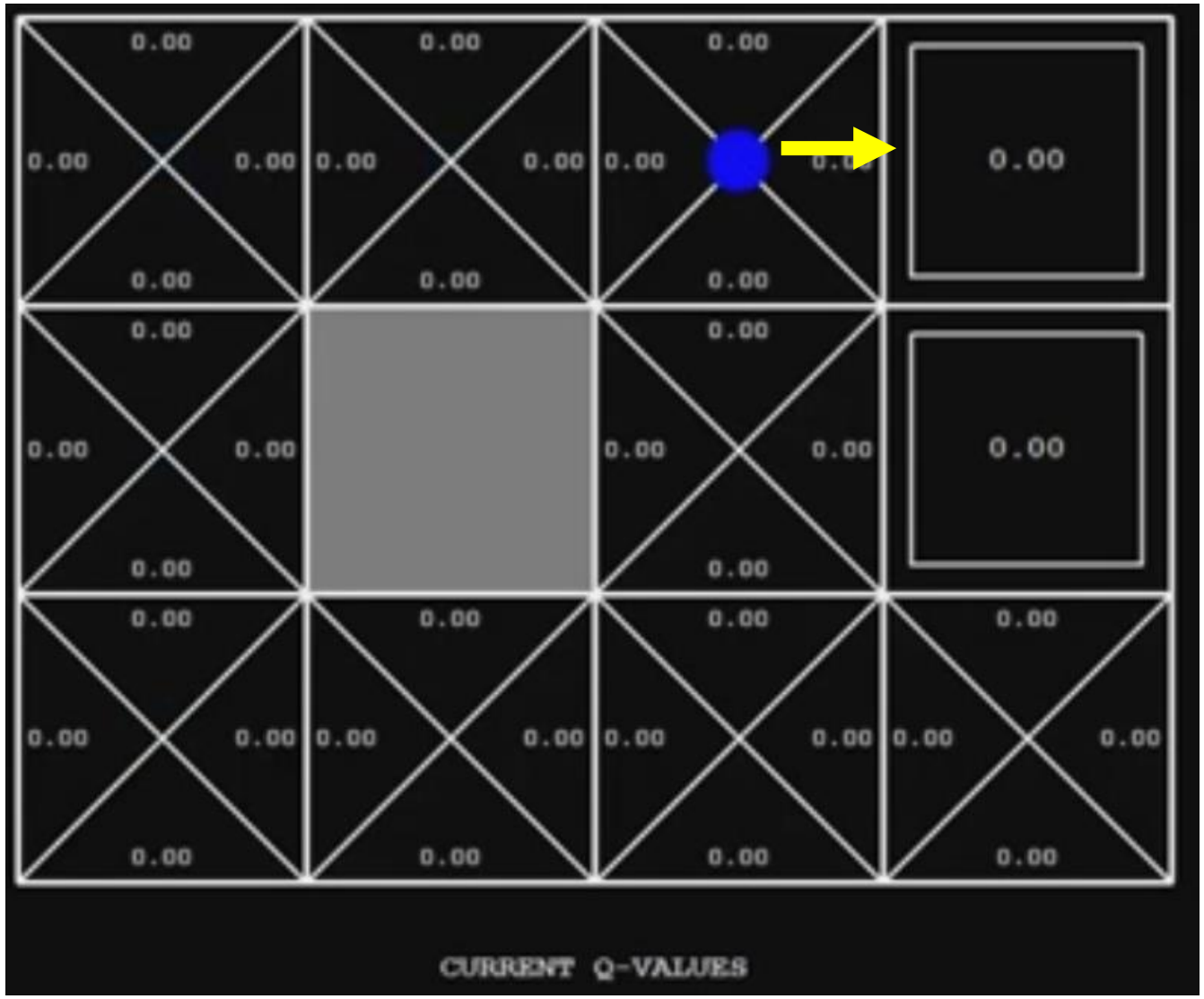
Q-Learning Algorithm (Cont'd)



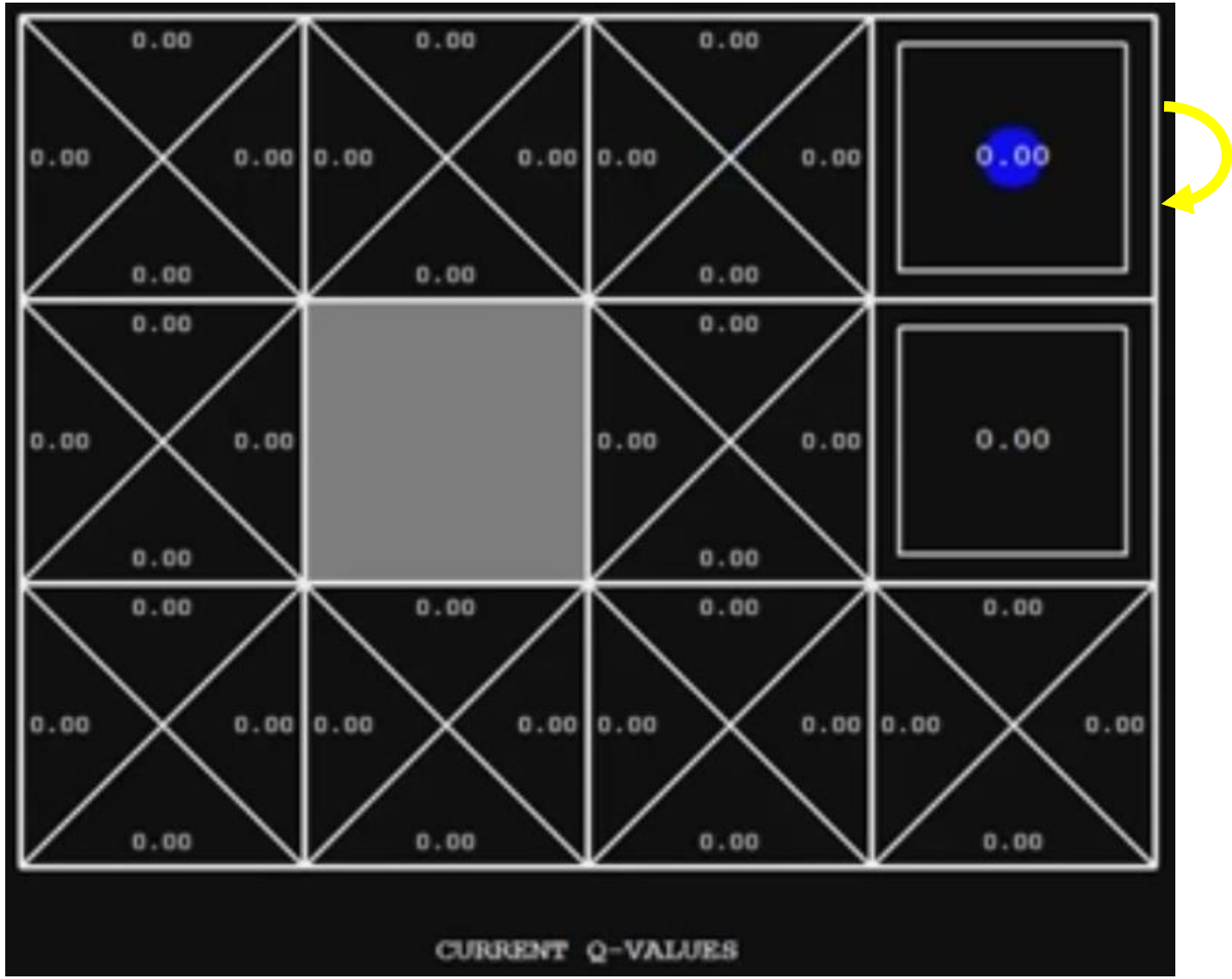
Q-Learning Algorithm (Cont'd)



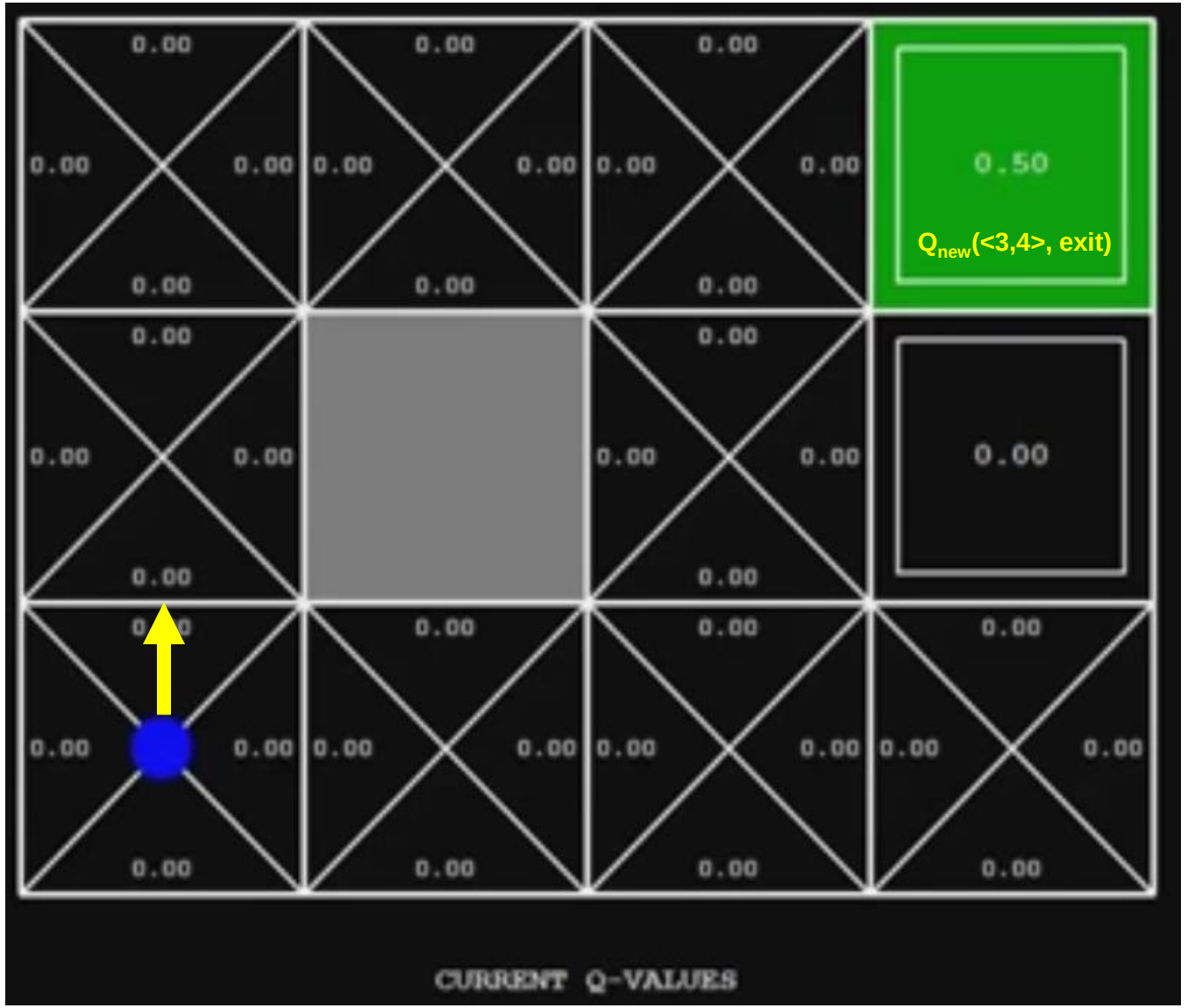
Q-Learning Algorithm (Cont'd)



Q-Learning Algorithm (Cont'd)



Q-Learning Algorithm (Cont'd)

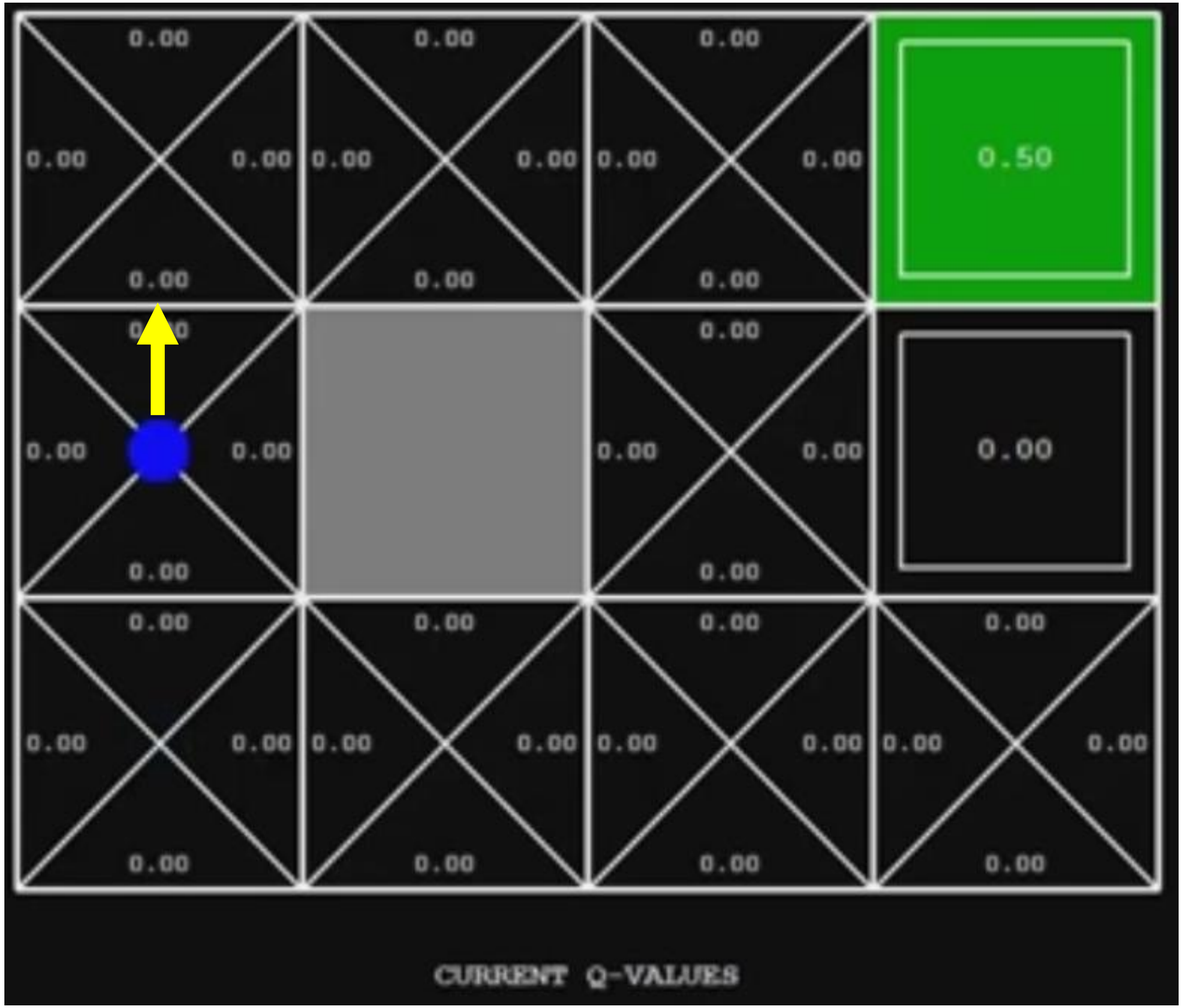


$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

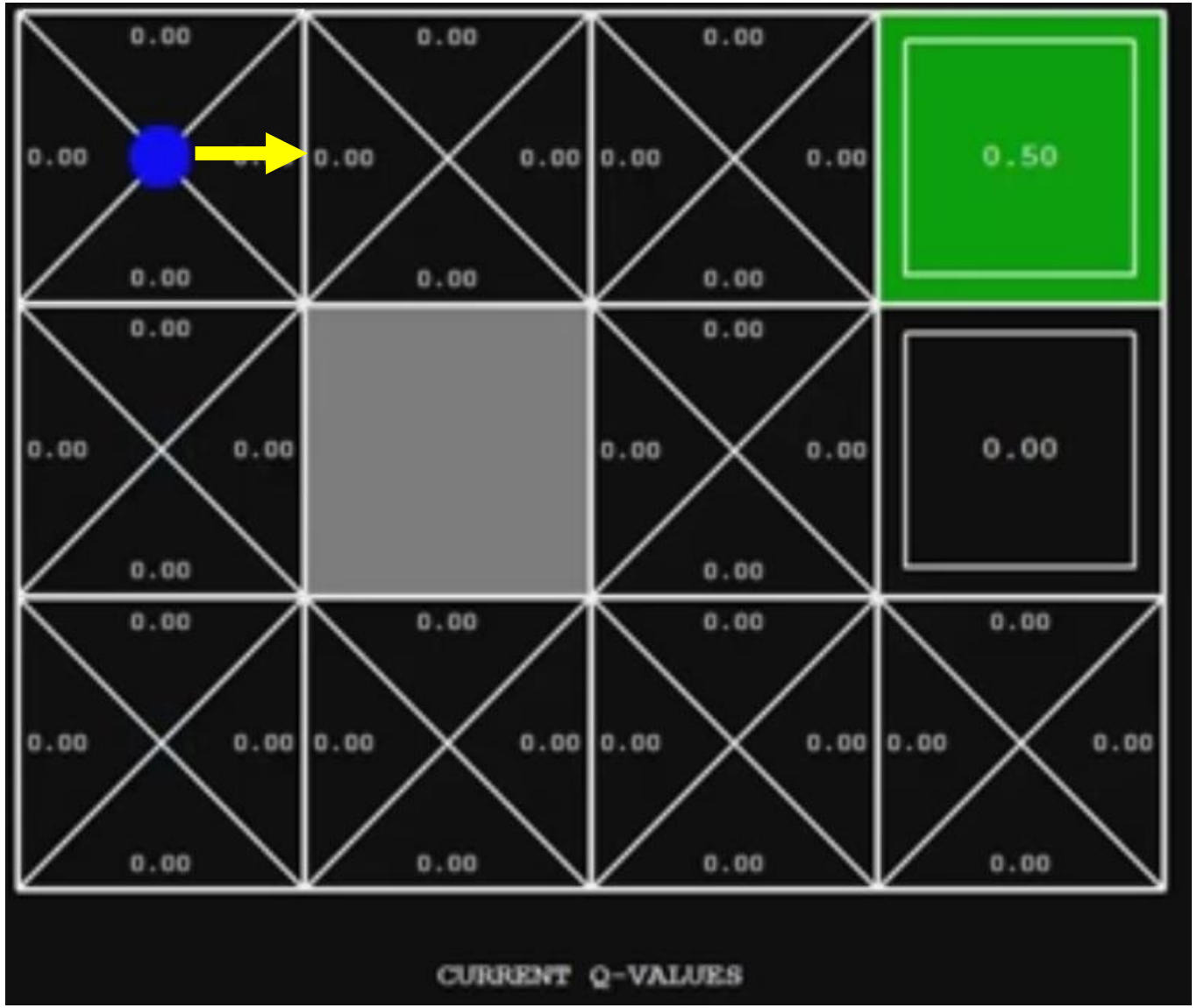
$= (1 - 0.5) \cdot Q_{old}(<3,4>, exit) + 0.5 * 1$

$Q_{new}(<3,4>, exit) = 0.5 \cdot 0 + 0.5 = 0.5$

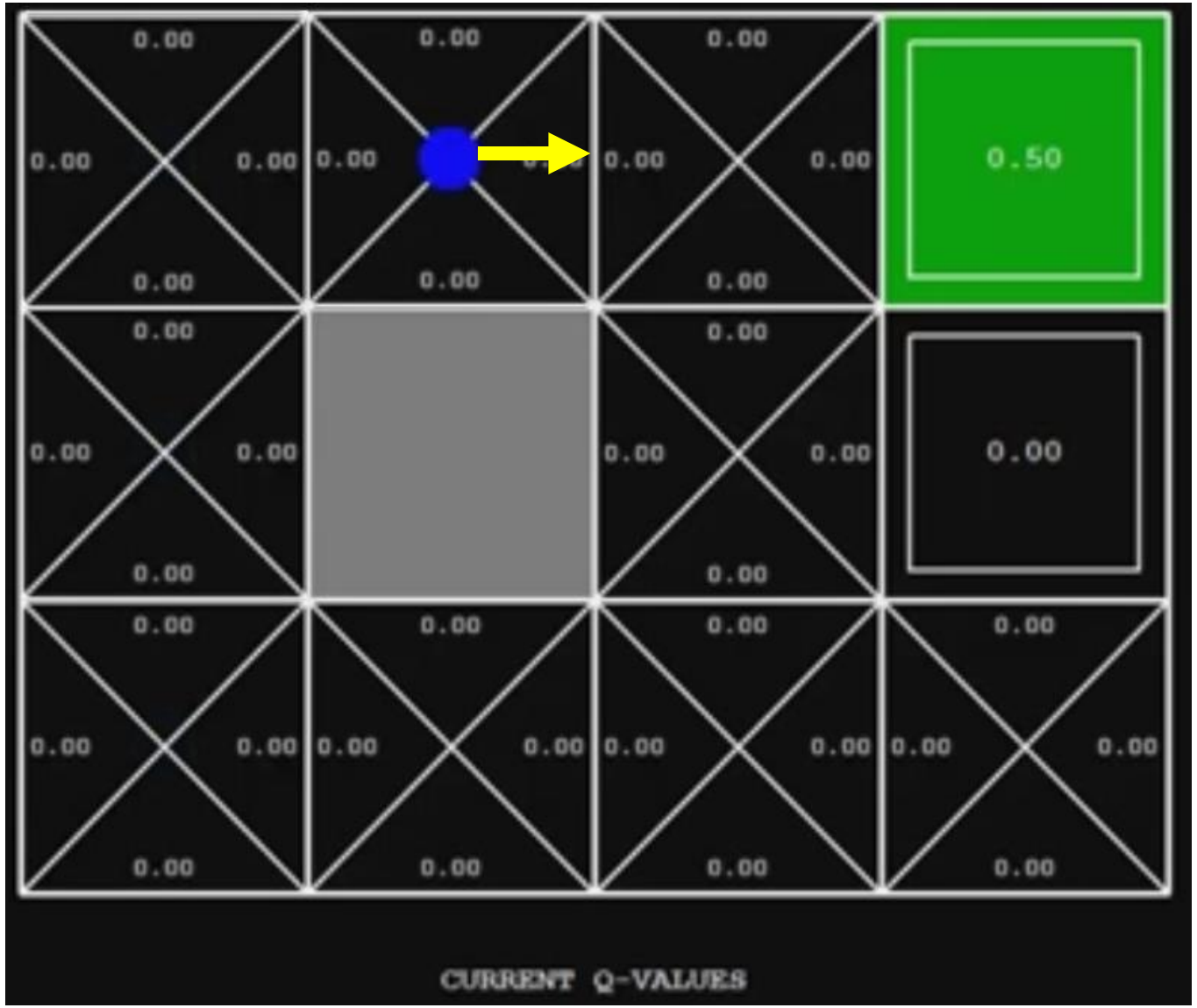
Q-Learning Algorithm (Cont'd)



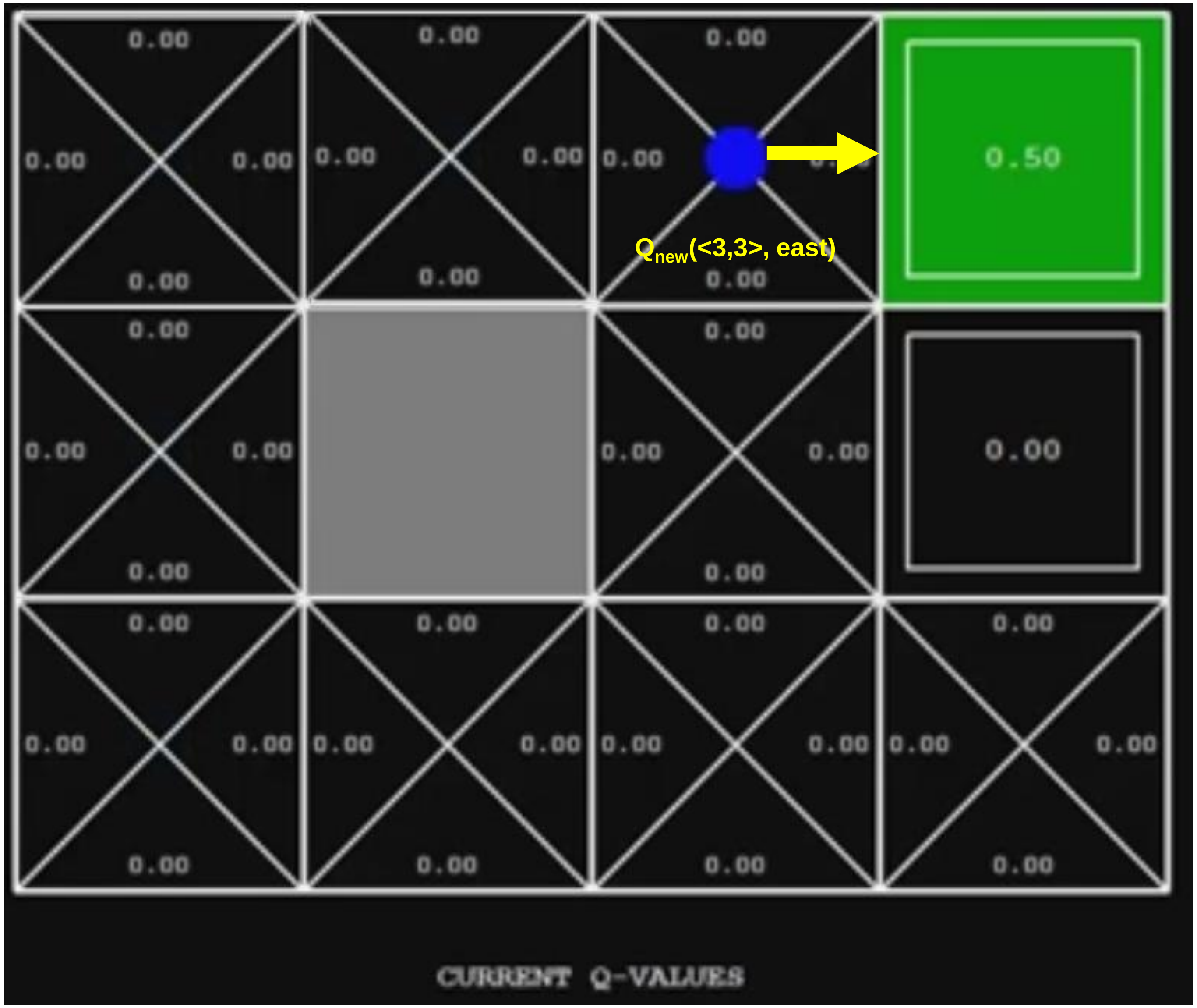
Q-Learning Algorithm (Cont'd)



Q-Learning Algorithm (Cont'd)



Q-Learning Algorithm (Cont'd)

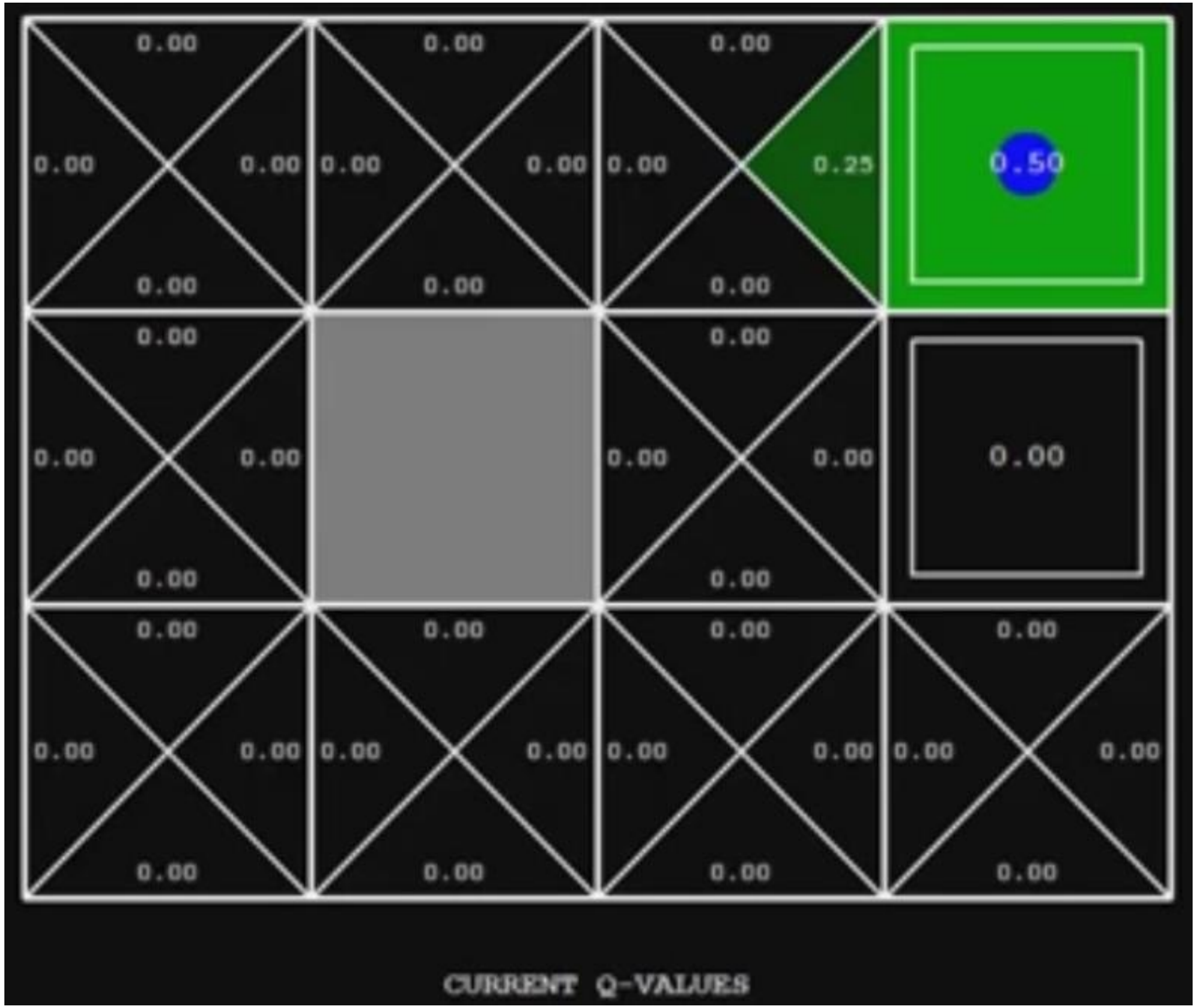


$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$

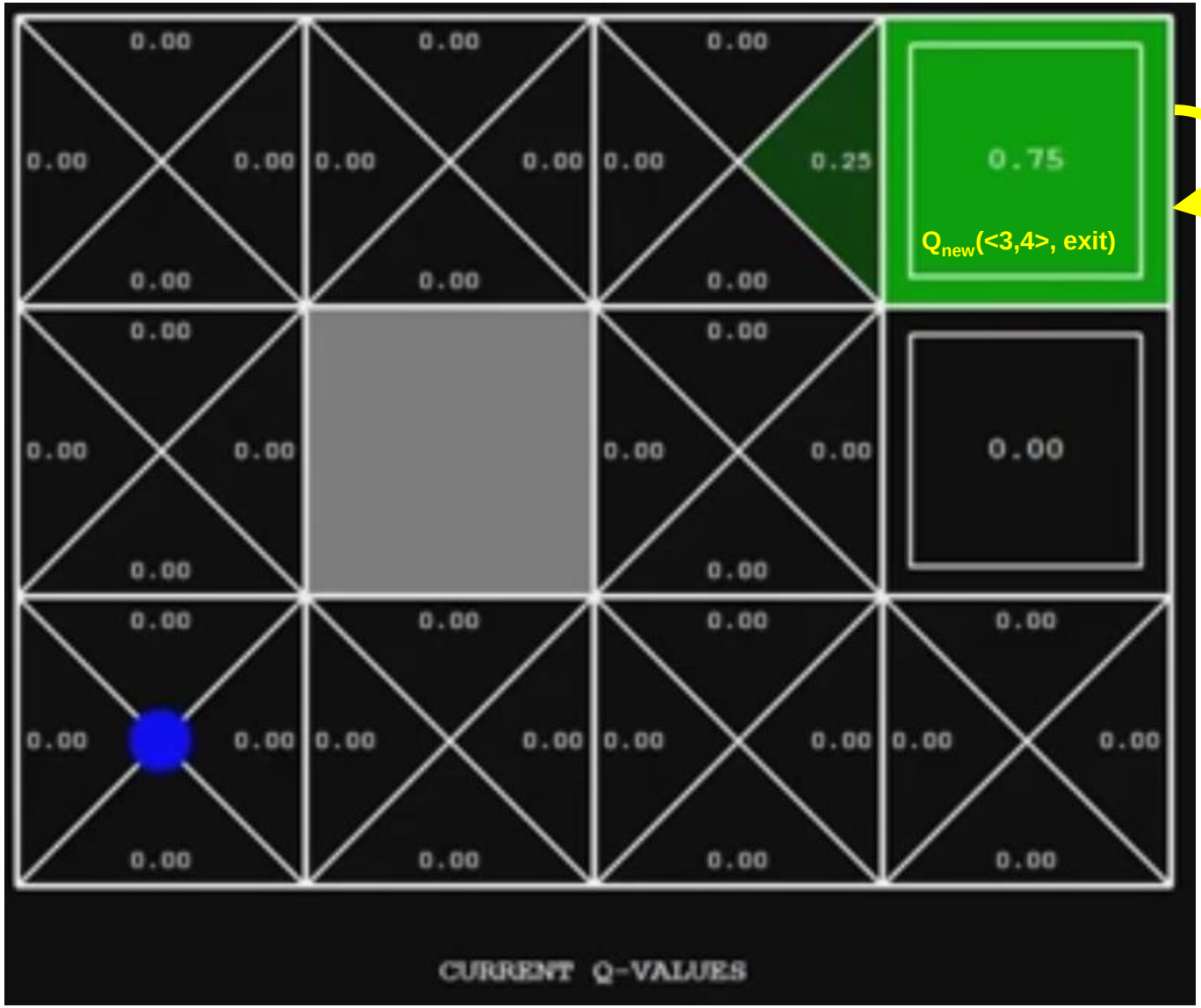
= (1 – 0.5) · Q_{old}(<3,3>, east) + 0.5 * 0.5

Q_{new}(<3,3>, east) = 0.5 · 0 + 0.25 = 0.25

Q-Learning Algorithm (Cont’d)



Q-Learning Algorithm (Cont'd)

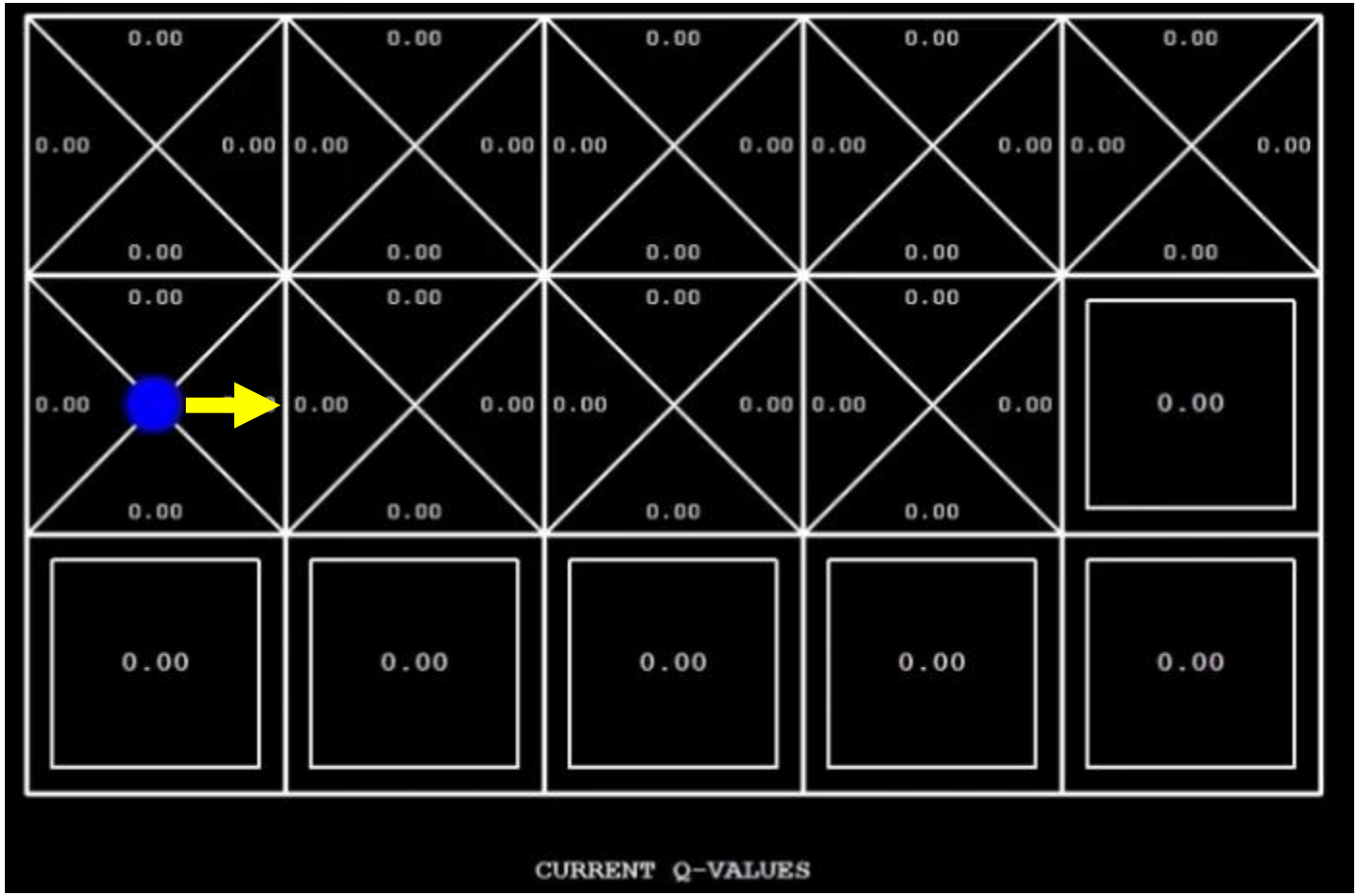


$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$

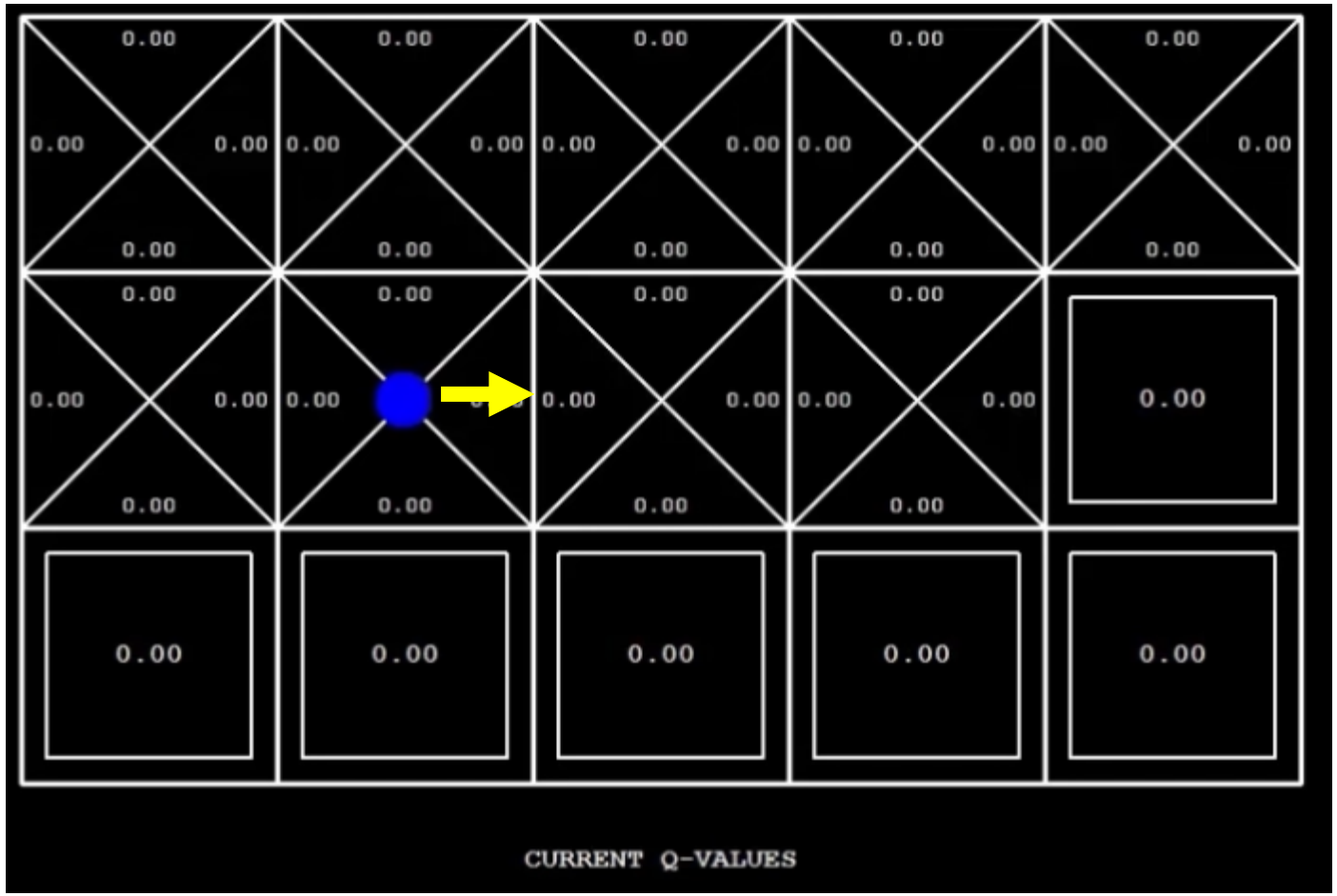
$= (1 - 0.5) \cdot Q_{old}(\langle 3,4\rangle, exit) + 0.5 * 1$

$Q_{new}(\langle 3,4\rangle, exit) = 0.5 \cdot 0.5 + 0.5 = 0.75$

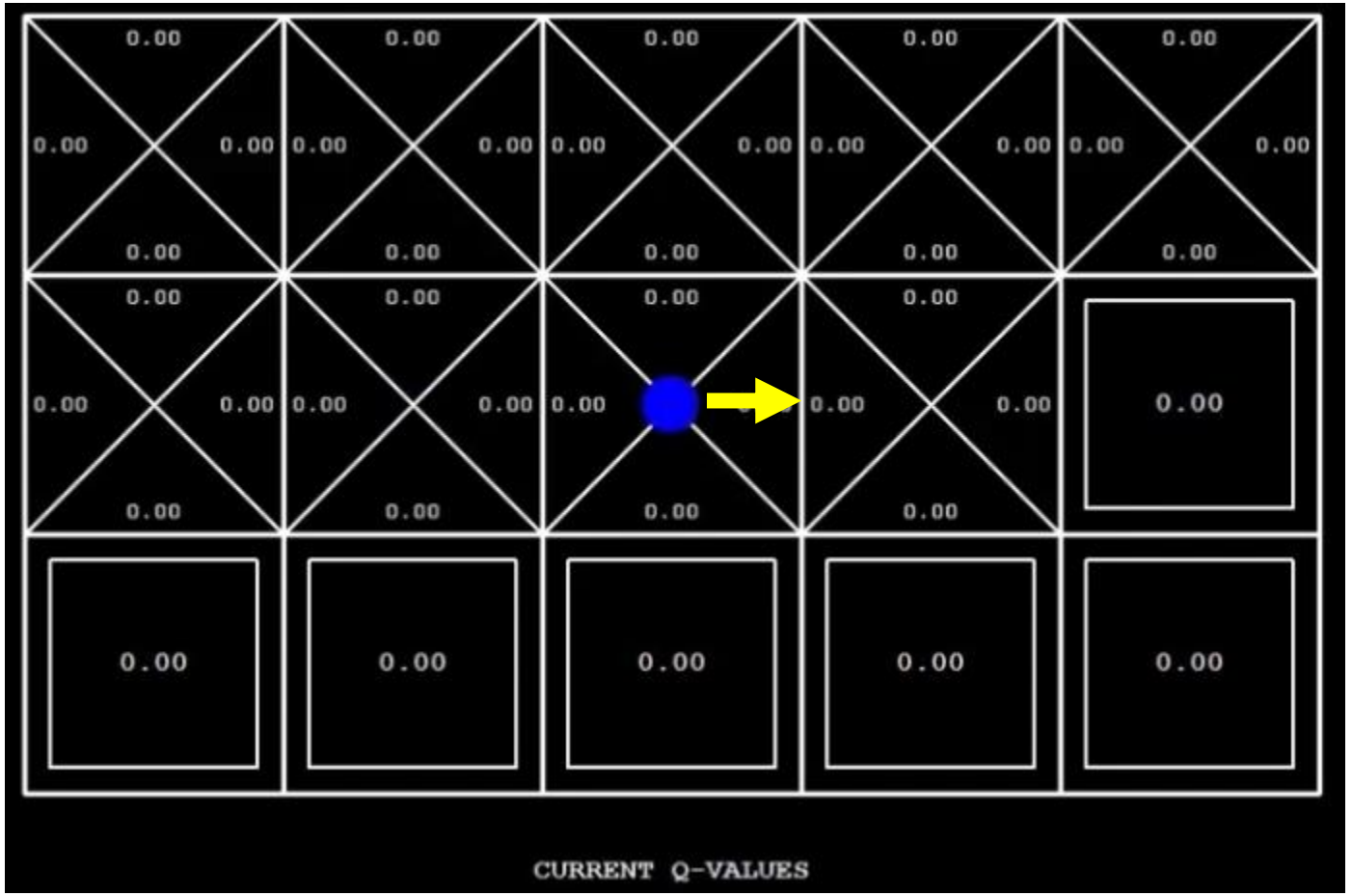
Q-Learning Algorithm: Another Example



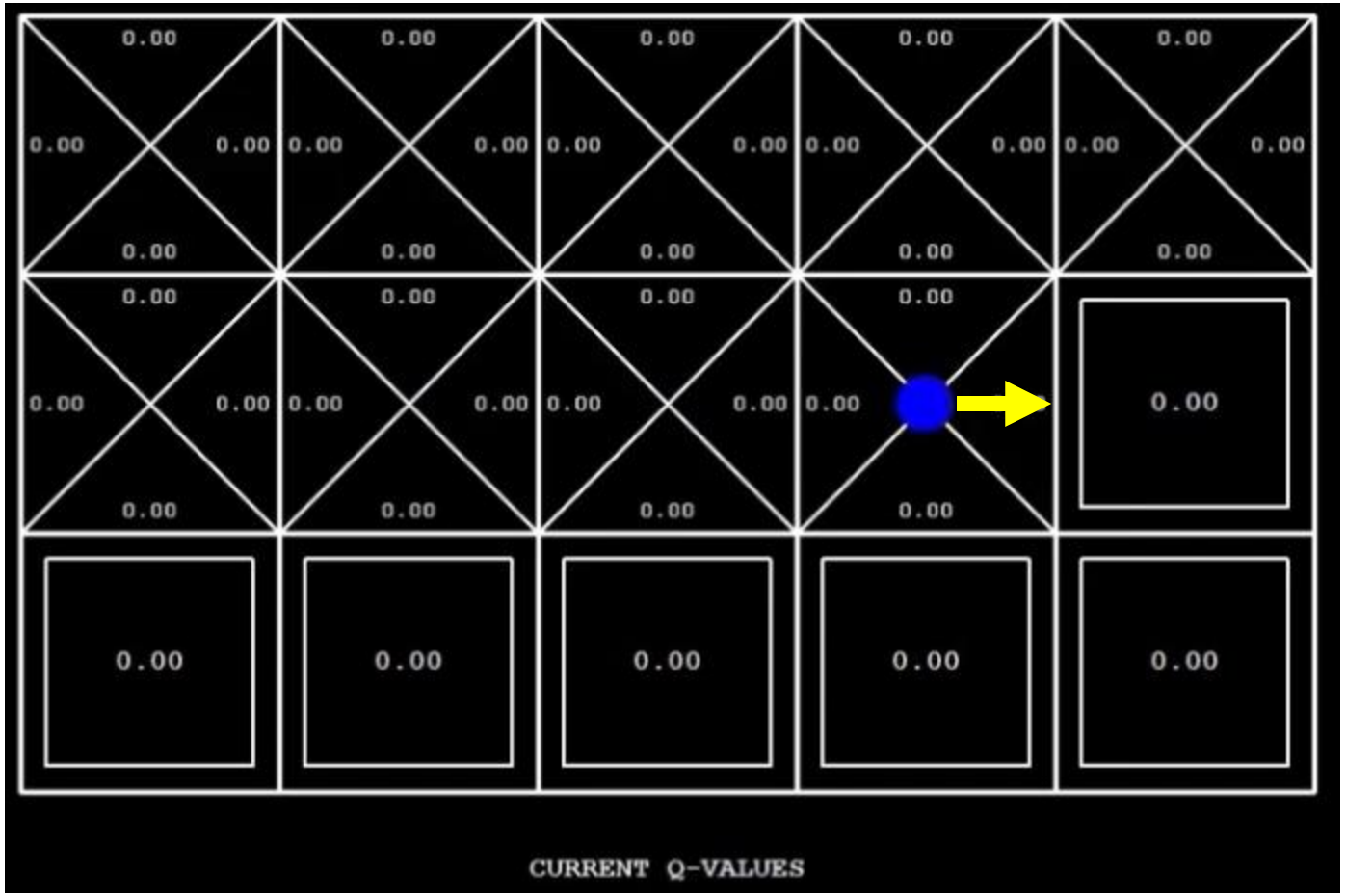
Q-Learning Algorithm: Another Example (Cont'd)



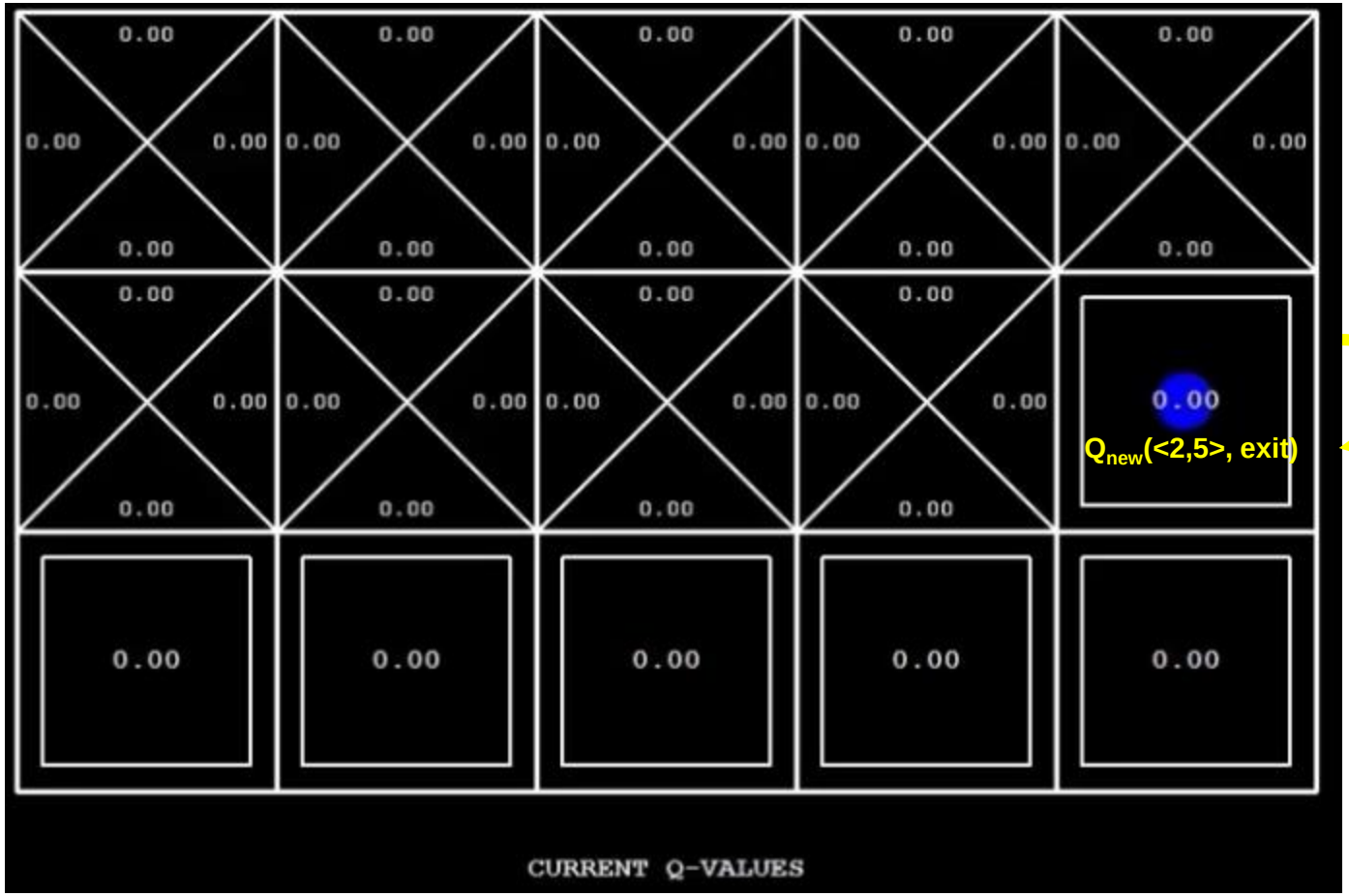
Q-Learning Algorithm: Another Example (Cont'd)



Q-Learning Algorithm: Another Example (Cont'd)



Q-Learning Algorithm: Another Example (Cont'd)



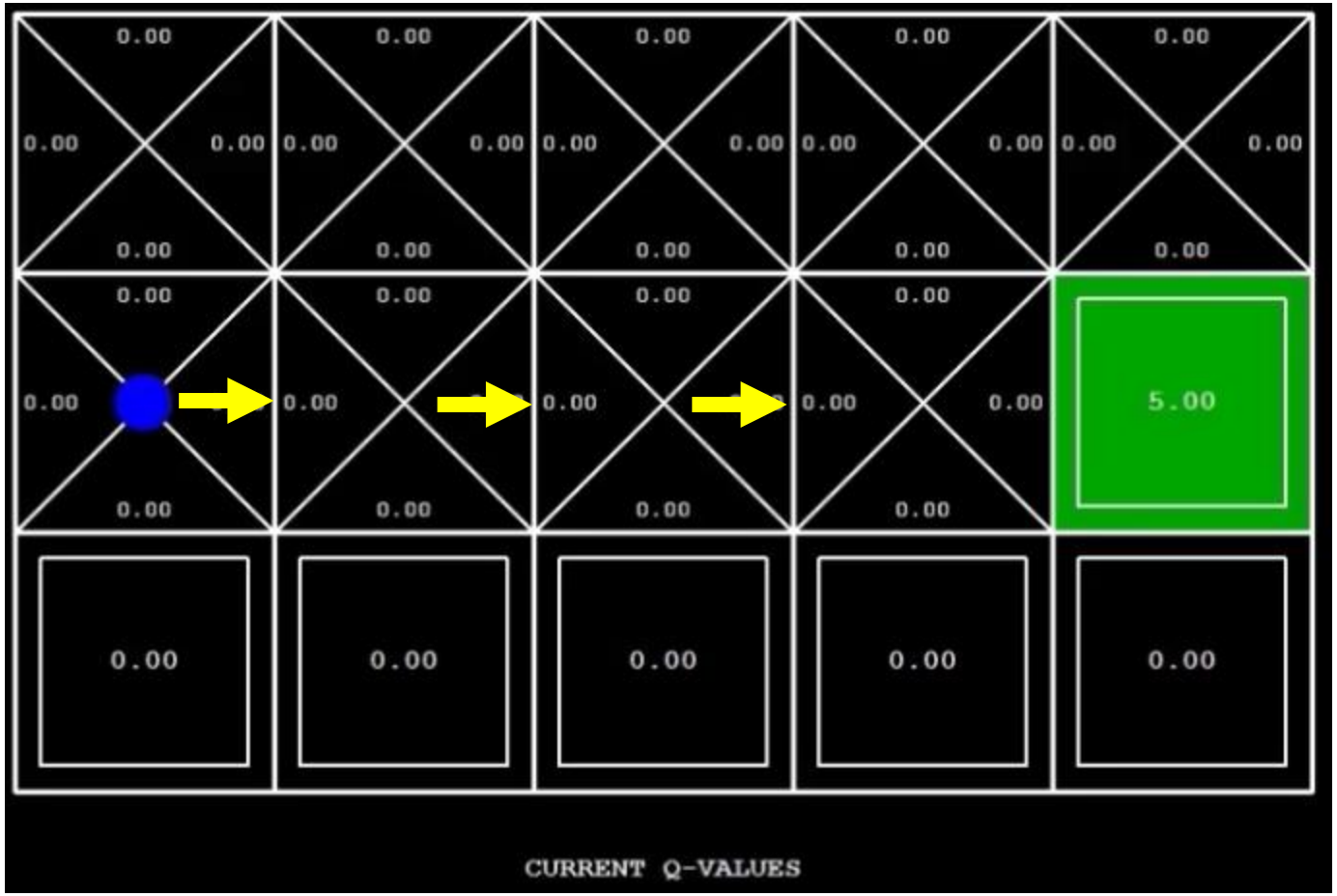
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

Q_{new}(<2,5>, exit)

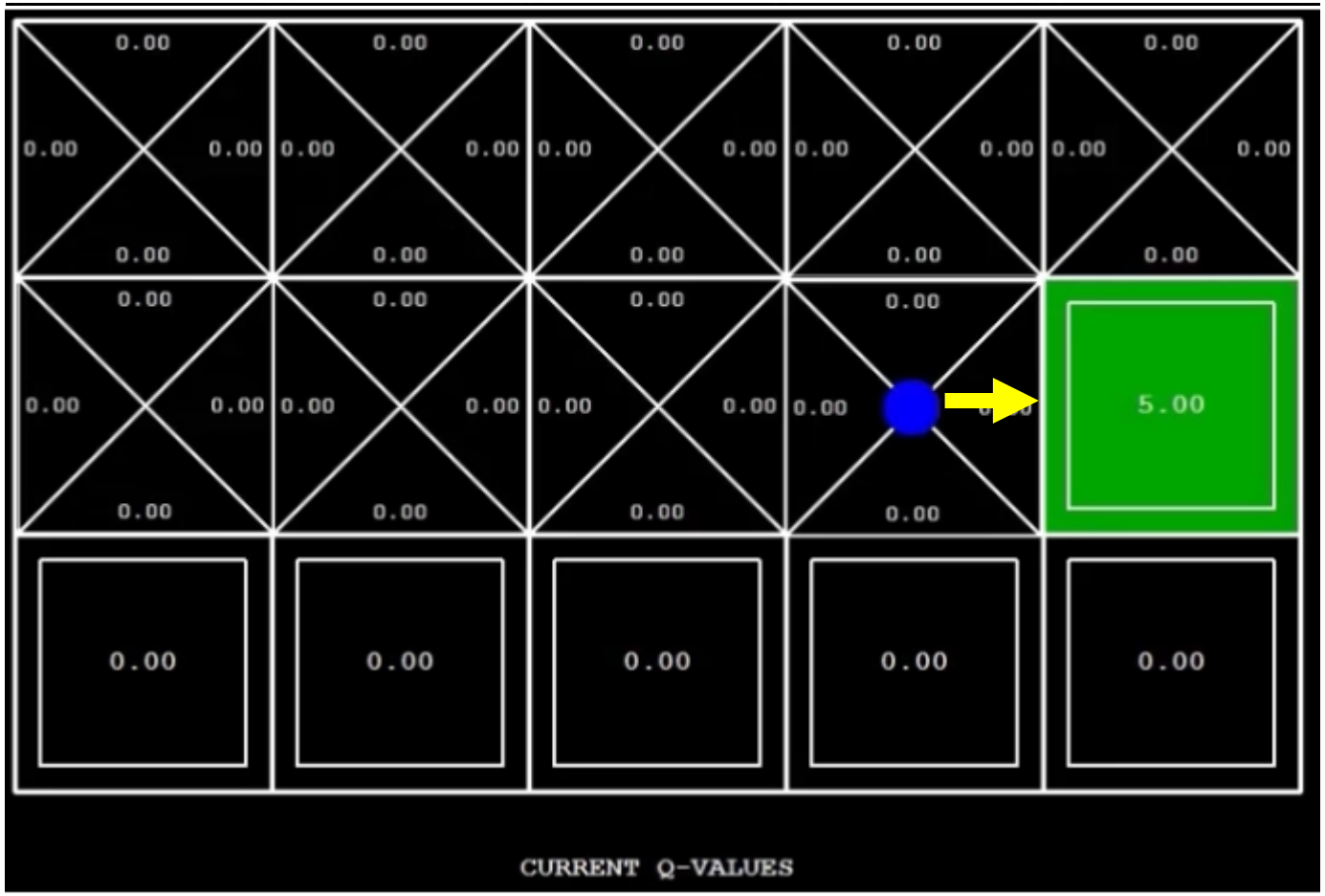
= (1 – 0.5) · Q_{old}(<2, 5>, exit) + 0.5 * 10

Q_{new}(<2, 5>, exit) = 0.5 · 0 + 5 = 5

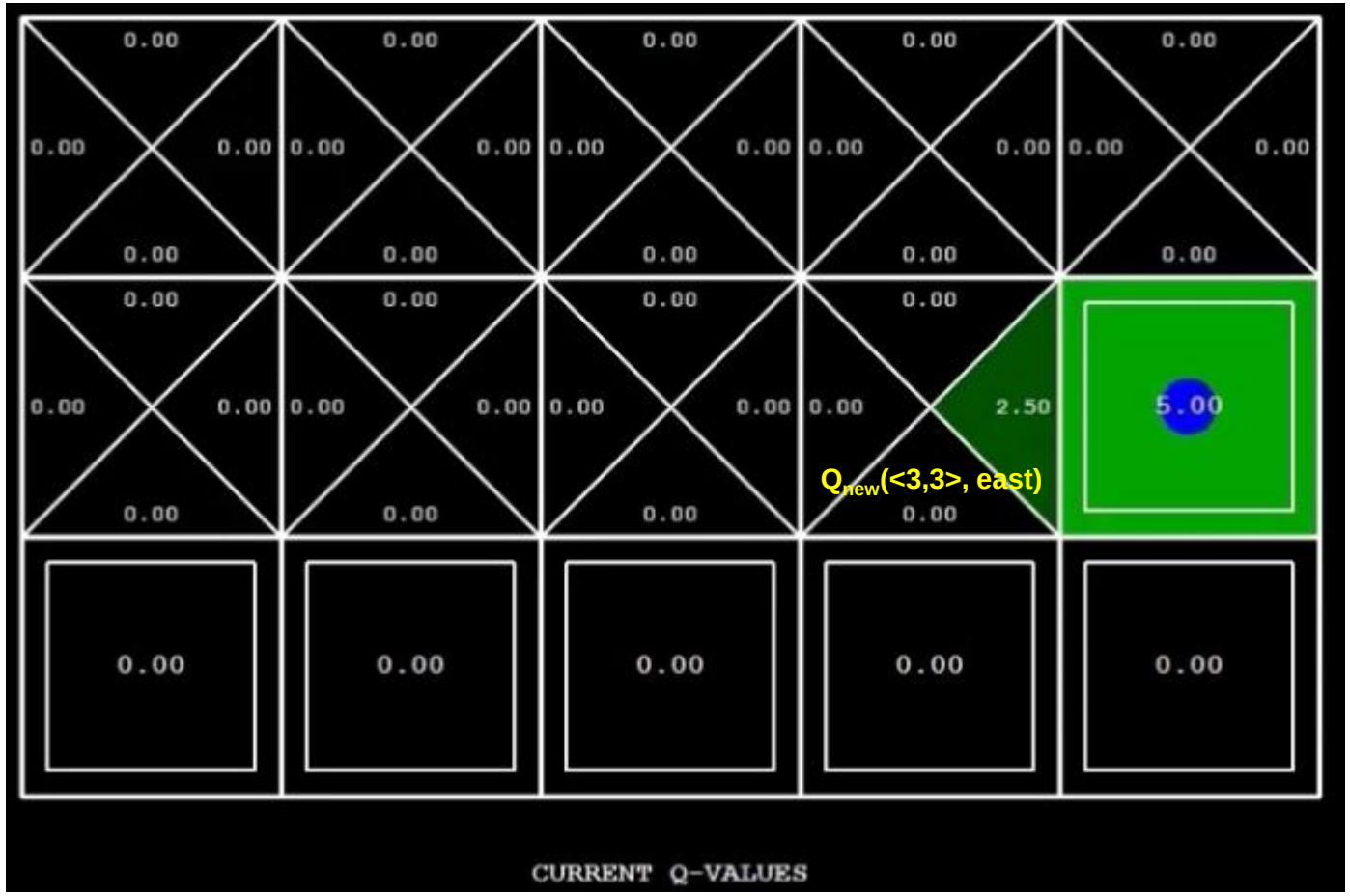
Q-Learning Algorithm: Another Example (Cont'd)



Q-Learning Algorithm: Another Example (Cont'd)



Q-Learning Algorithm: Another Example (Cont'd)

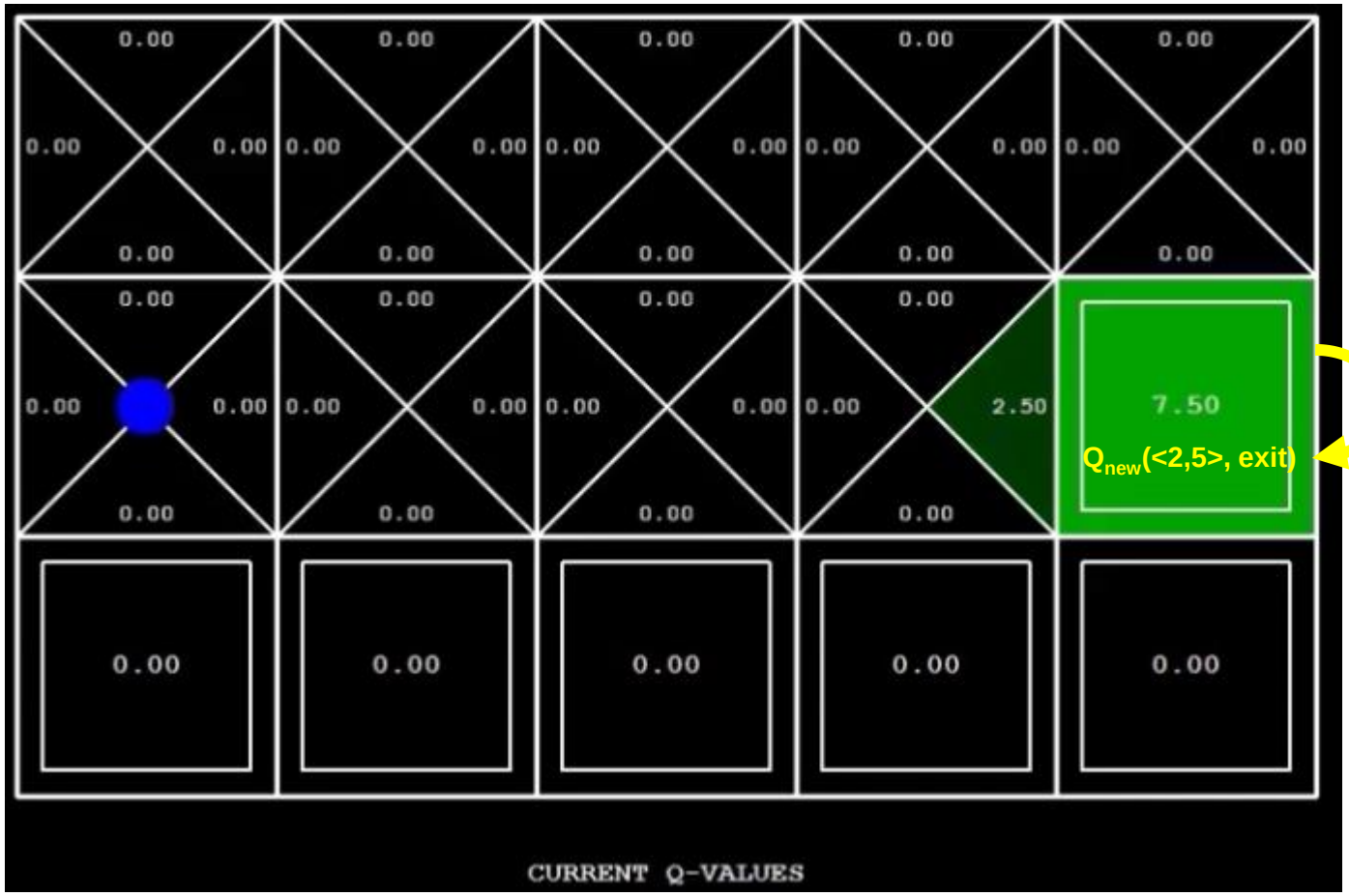


$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

Q_{new}(<3,3>, east) = (1 – 0.5) · Q_{old}(<3,3>, east) + 0.5 * 5

Q_{new}(<3,3>, east) = 0.5 · 0 + 2.5 = 2.5

Q-Learning Algorithm: Another Example (Cont'd)

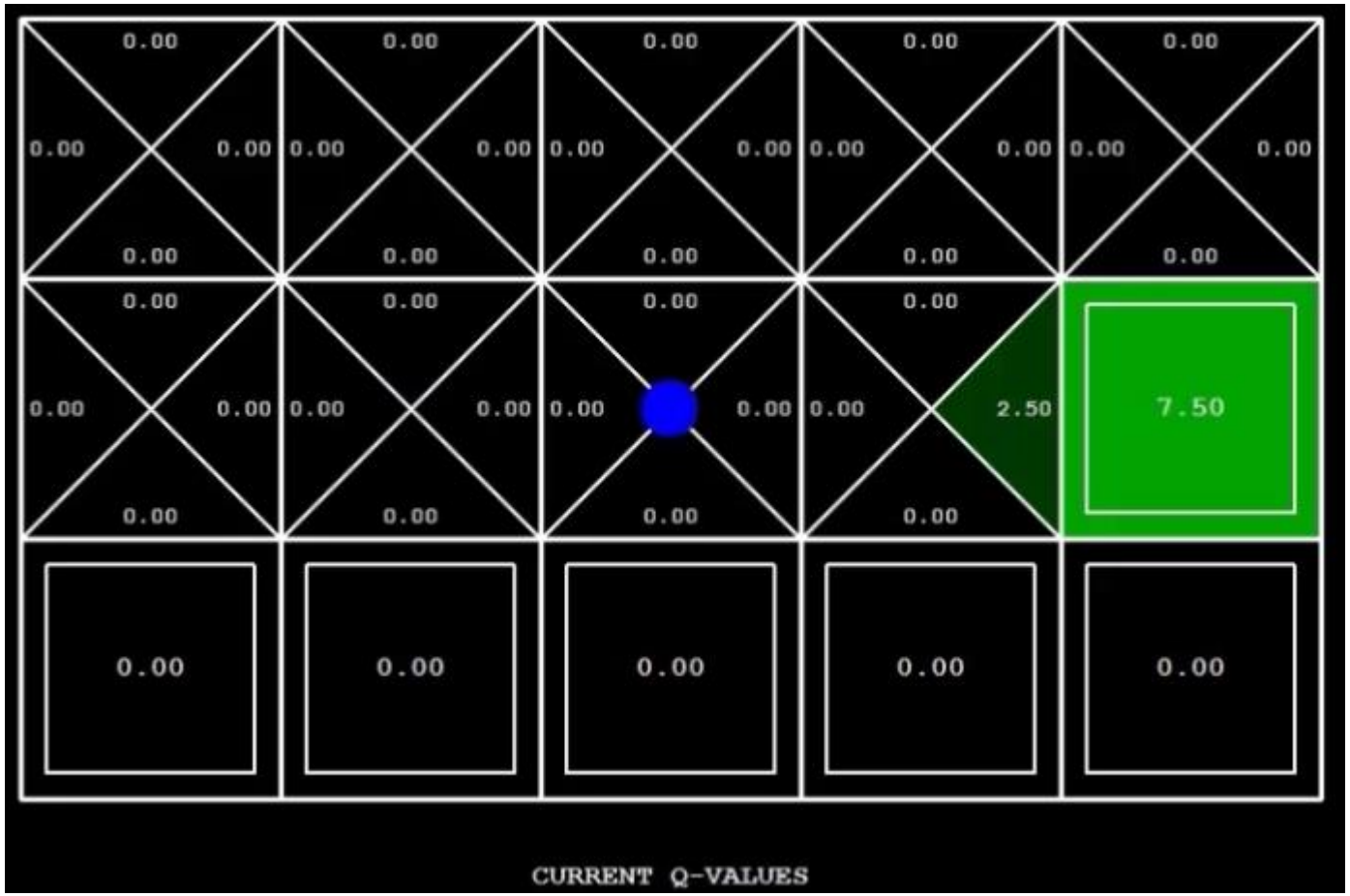


$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

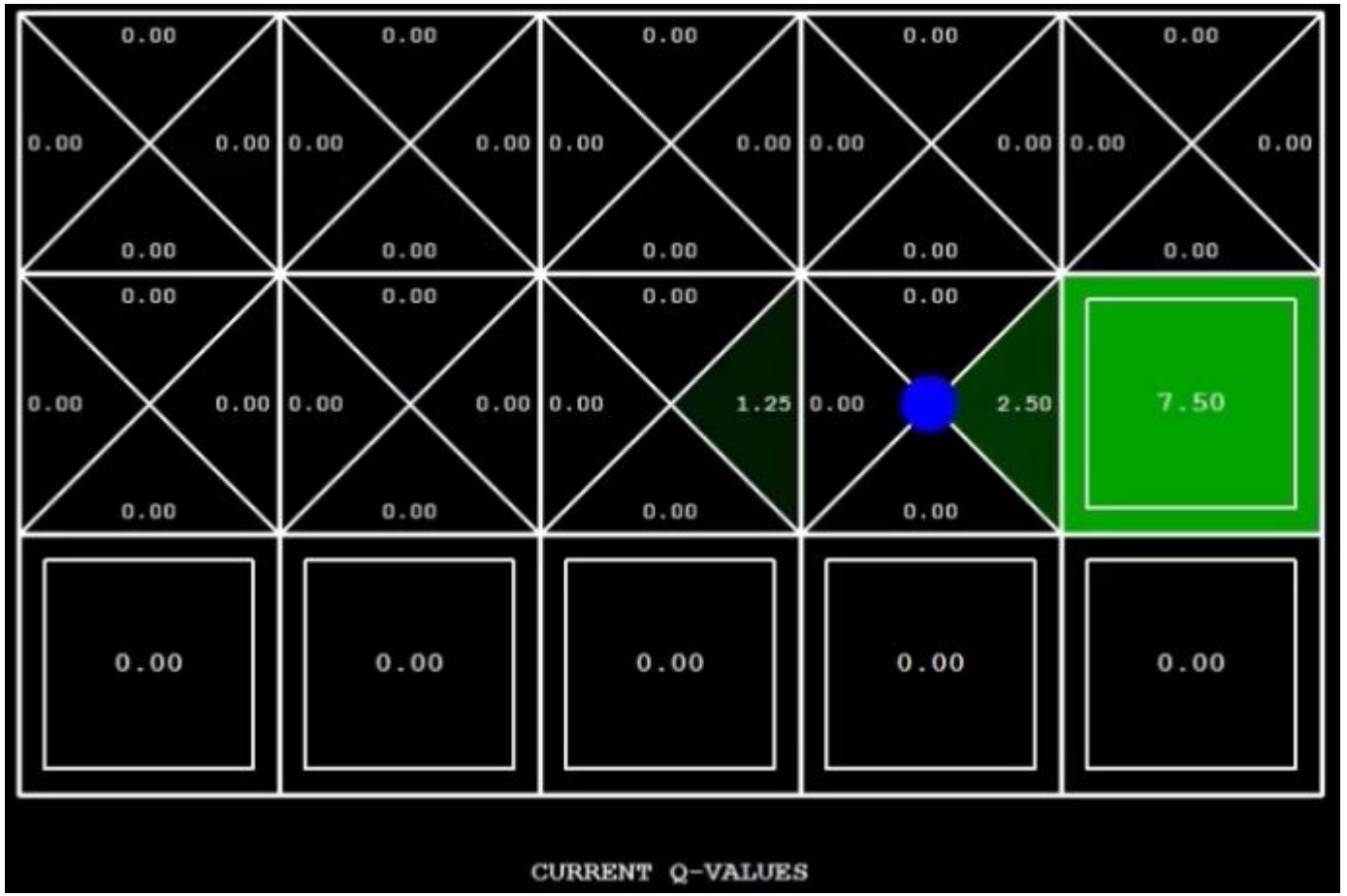
$Q_{new}(<2,5>, exit) = (1 - 0.5) \cdot Q_{old}(<2,5>, exit) + 0.5 * 10$

$Q_{new}(<2,5>, exit) = 0.5 \cdot 5 + 5 = 7.5$

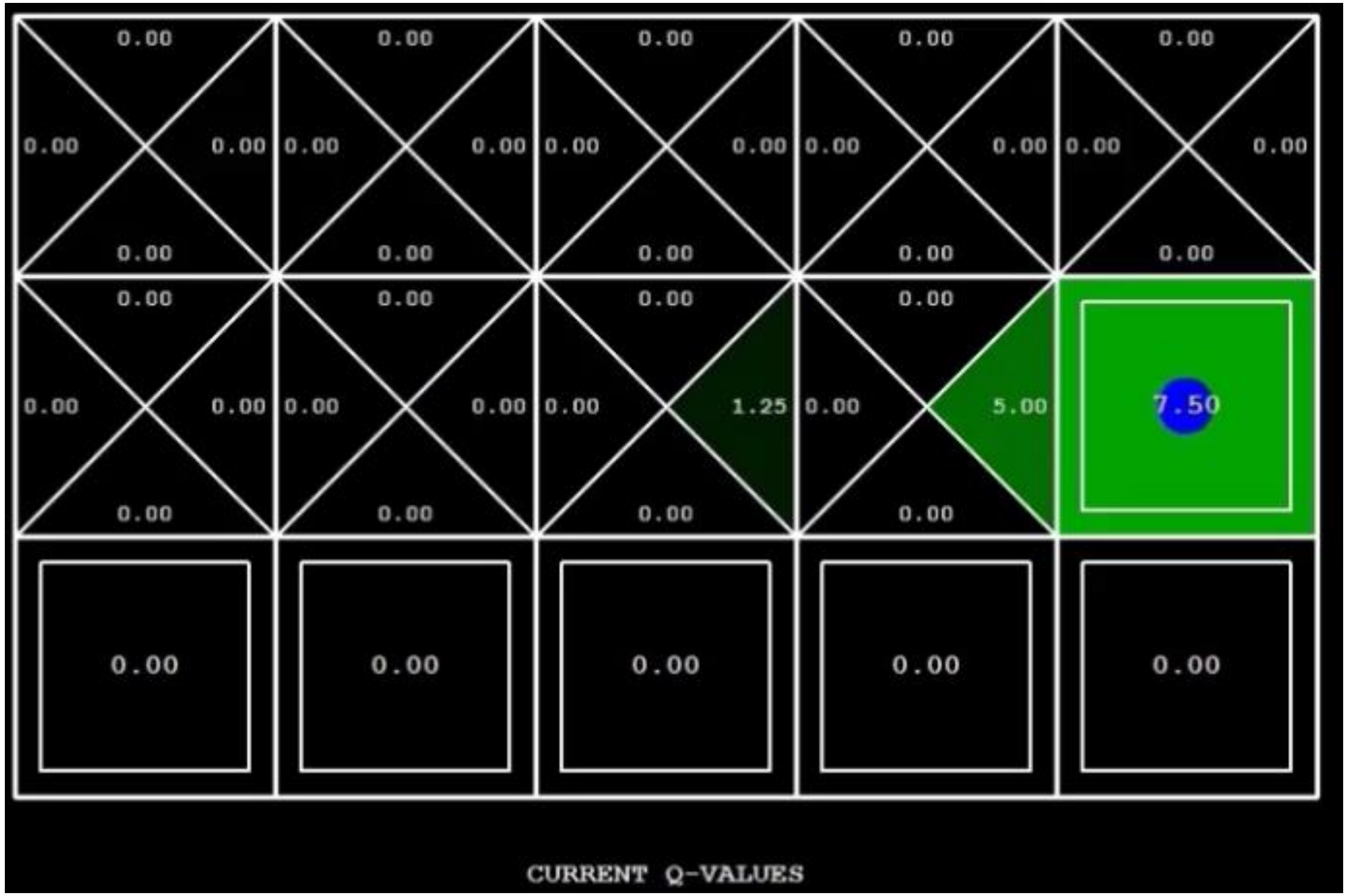
Q-Learning Algorithm: Another Example (Cont'd)



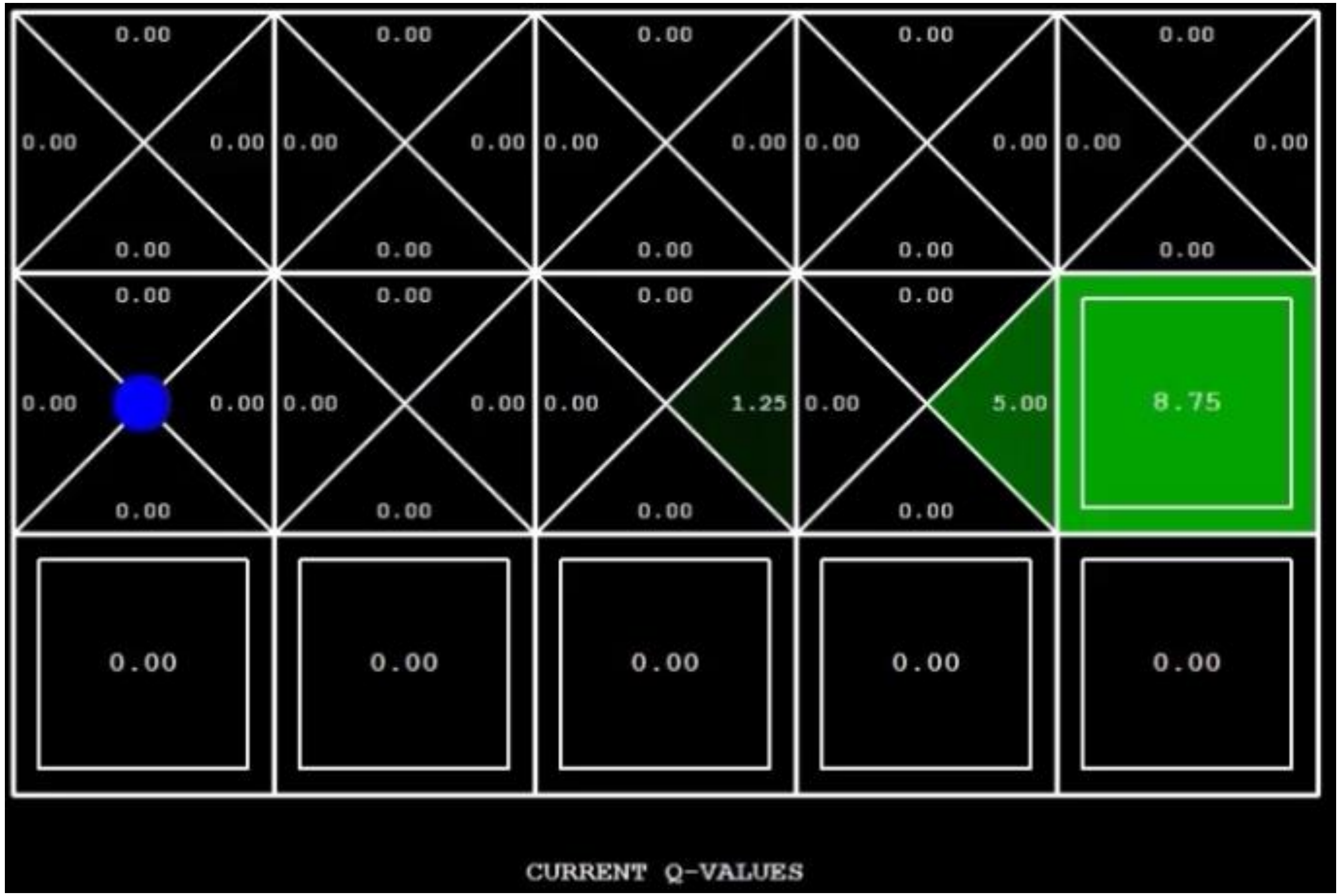
Q-Learning Algorithm: Another Example (Cont'd)



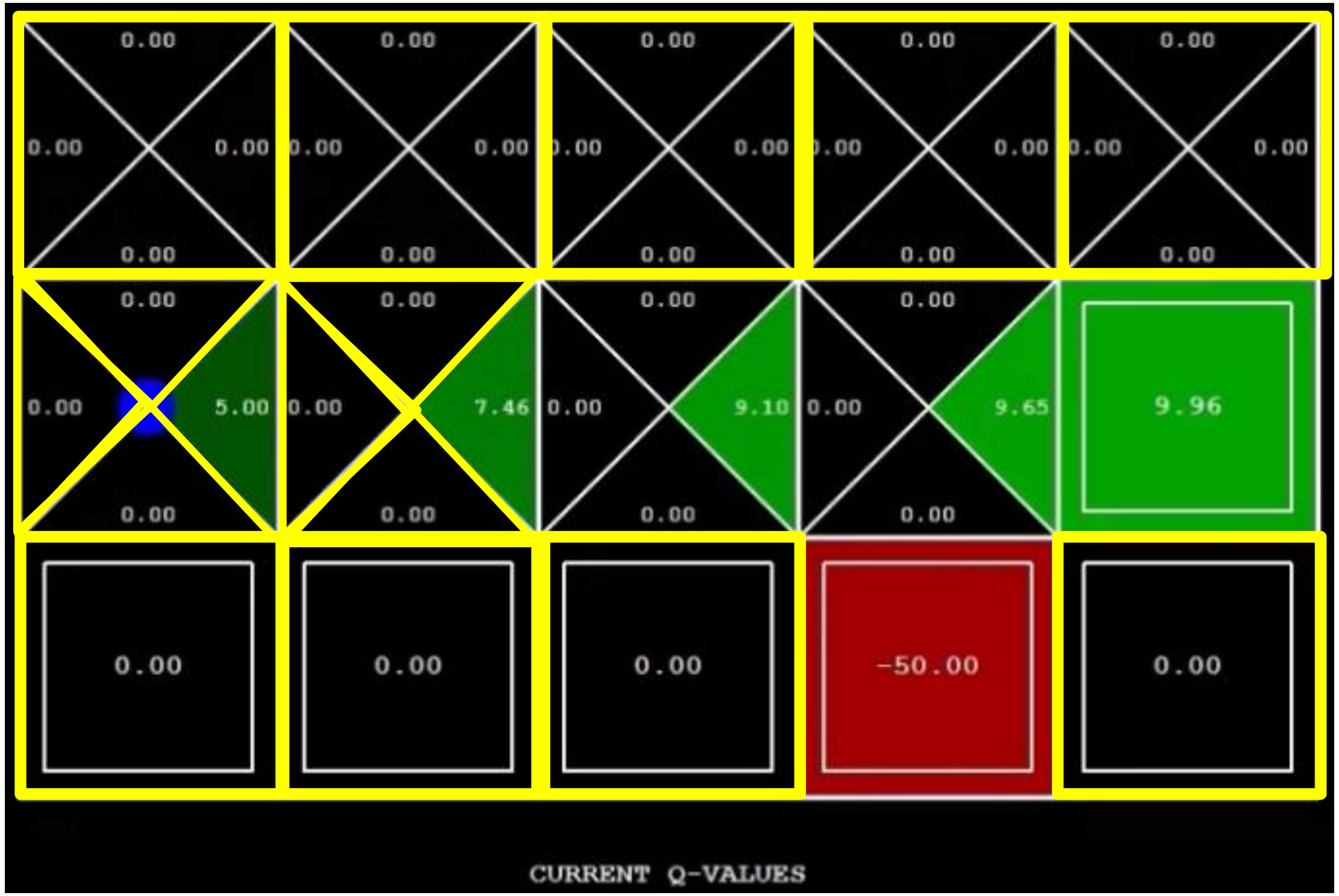
Q-Learning Algorithm: Another Example (Cont'd)



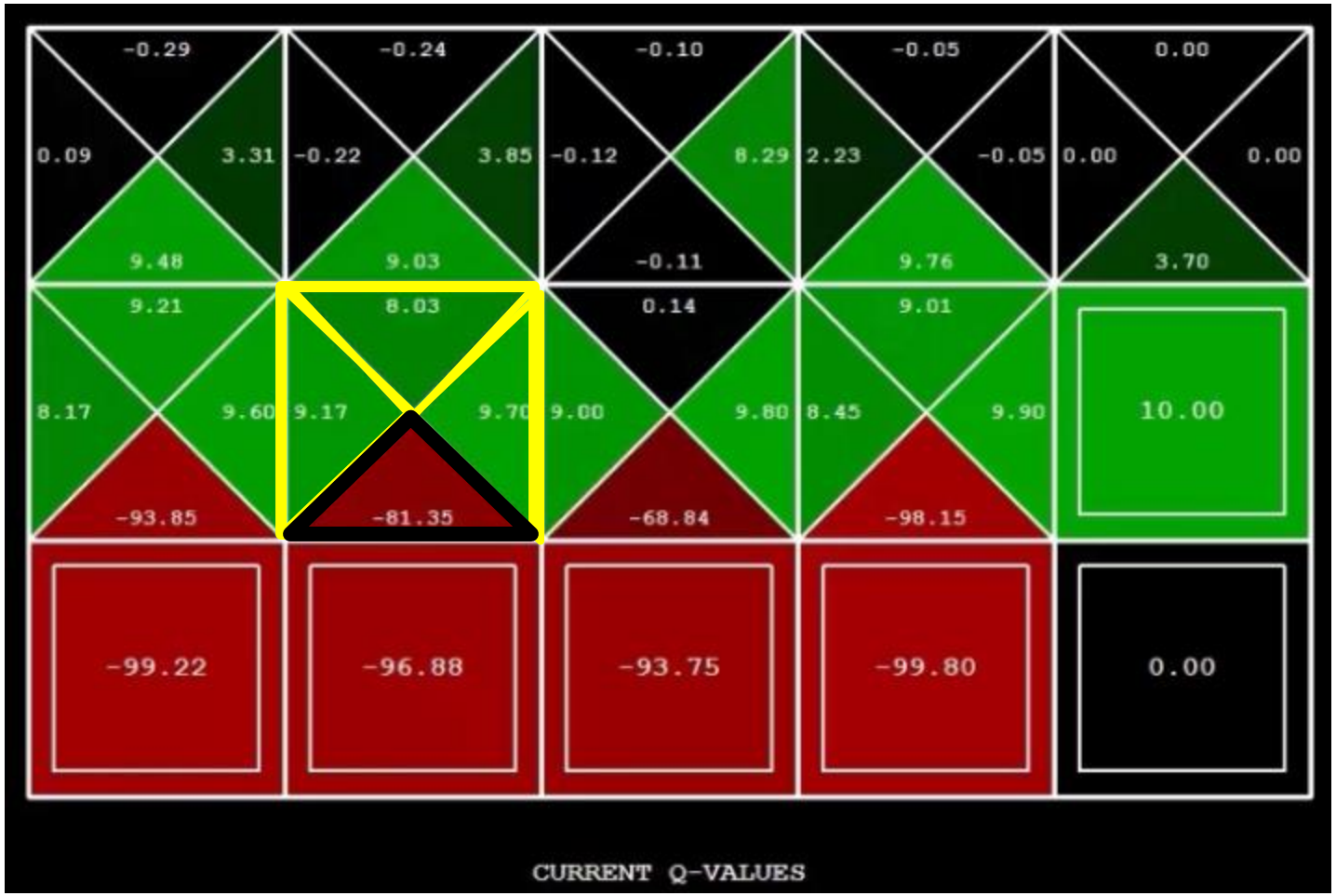
Q-Learning Algorithm: Another Example (Cont'd)



Q-Learning Algorithm: Another Example (Cont'd)

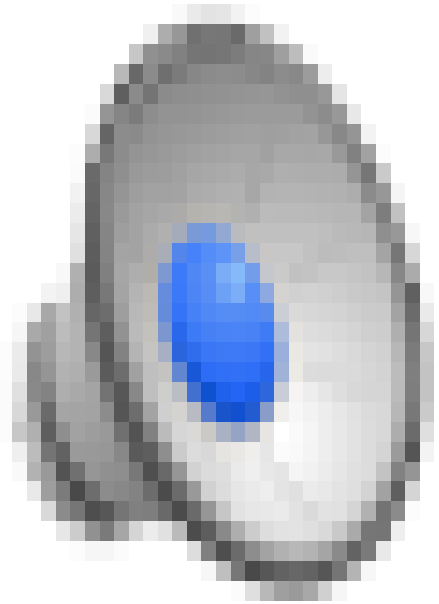


Q-Learning Algorithm: Another Example (Cont'd)

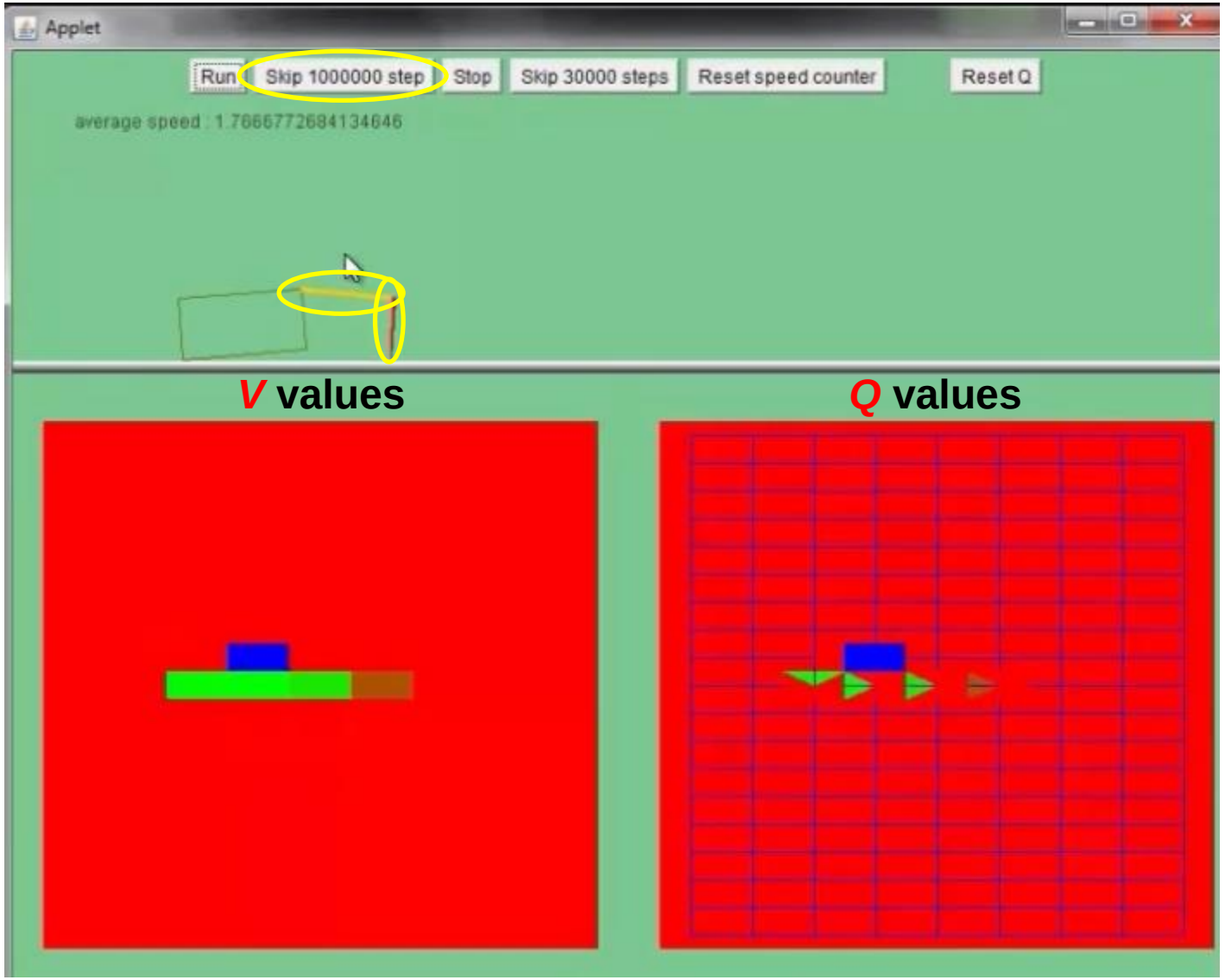


Note: The "goodness" of a state depends on its actions!

Demo of Q-Learning Algorithm (Gridworld)



Q-Learning for Robot Crawler

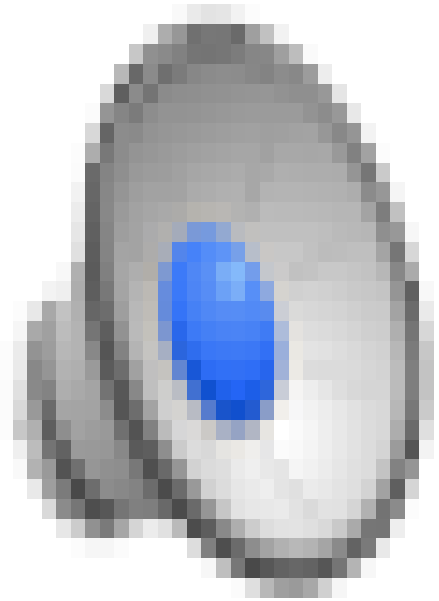


State: 2D state space

[Discretized]

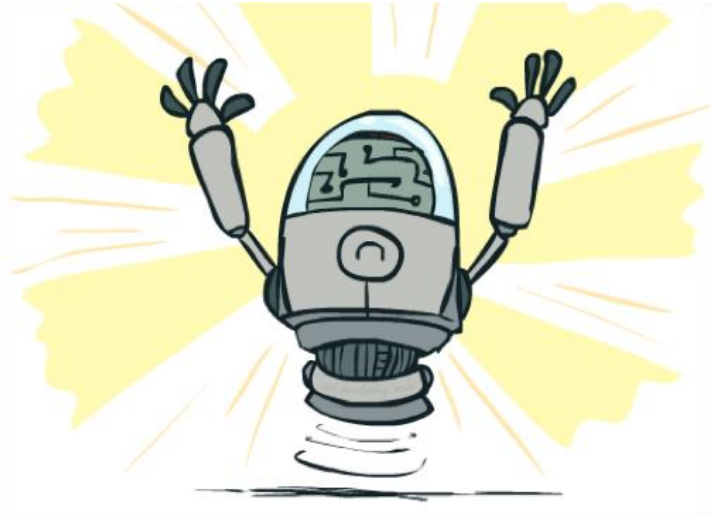
- **Horizontal axis of one joint angle**
- **Vertical axis of the other joint angle**
- **Policy followed 20% of the time**
- **Crawler acts randomly 80% of the time!**

Demo of Q-Learning Algorithm (Crawler)



Q-Learning Properties

- **Amazing result:** Q-learning converges to optimal policy -- even if you are acting suboptimally!



- This is called ***off-policy learning***

- **Caveats:**
  - You have to ***explore enough***
  - You have to eventually make the ***learning rate small enough***
  - … but ***not decrease it too quickly!***
  - Basically, in the limit, it ***does not matter how you select actions***!

References

Lecture slides adapted from:

UC Berkeley CS188 Intro to AI,

http://ai.berkeley.edu/home.html