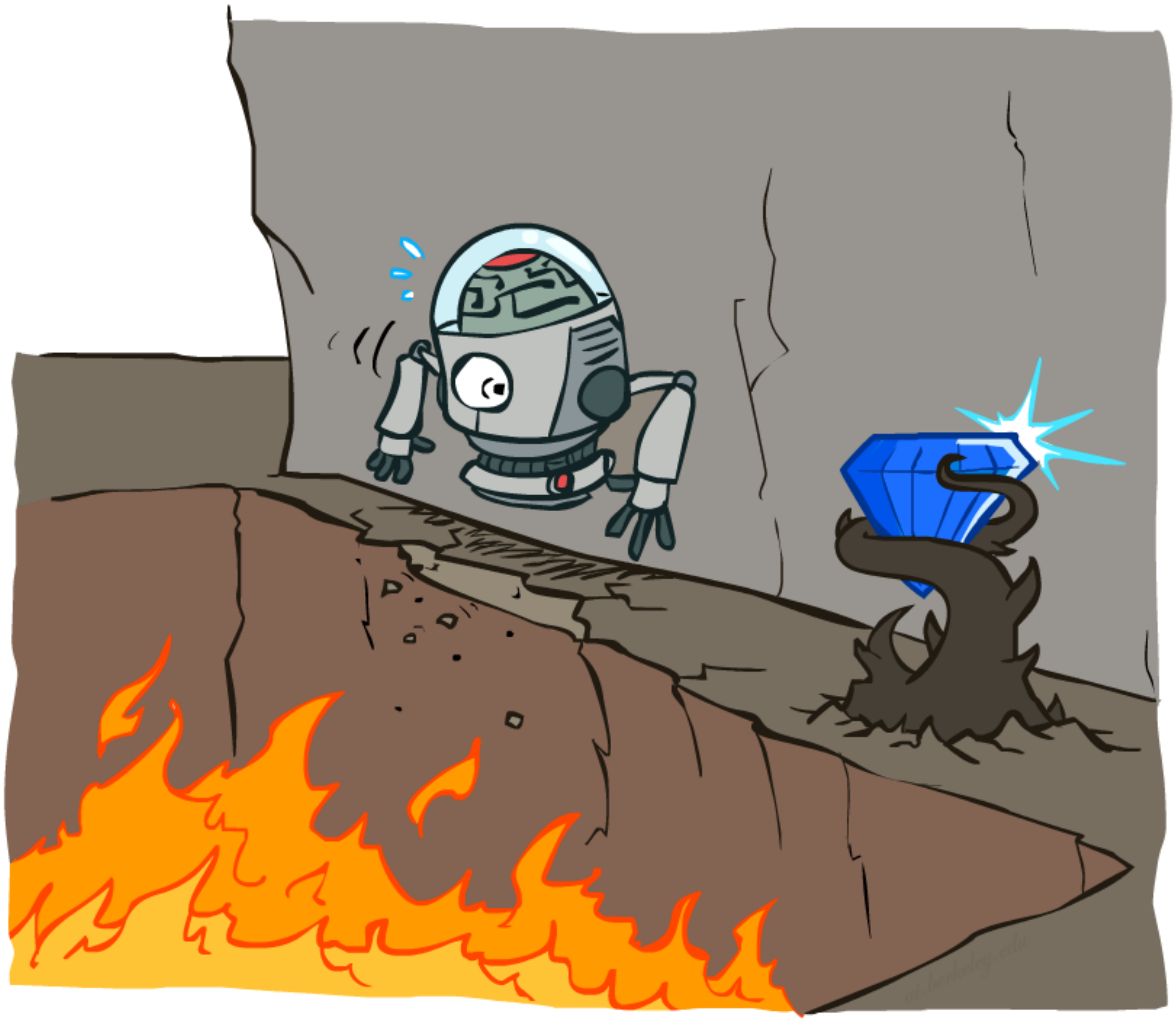
Markov Decision Processes

Chris Amato
Northeastern University

Some images and slides are used from: Rob Platt, CS188 UC Berkeley, AIMA

Stochastic domains



Sequential decision making

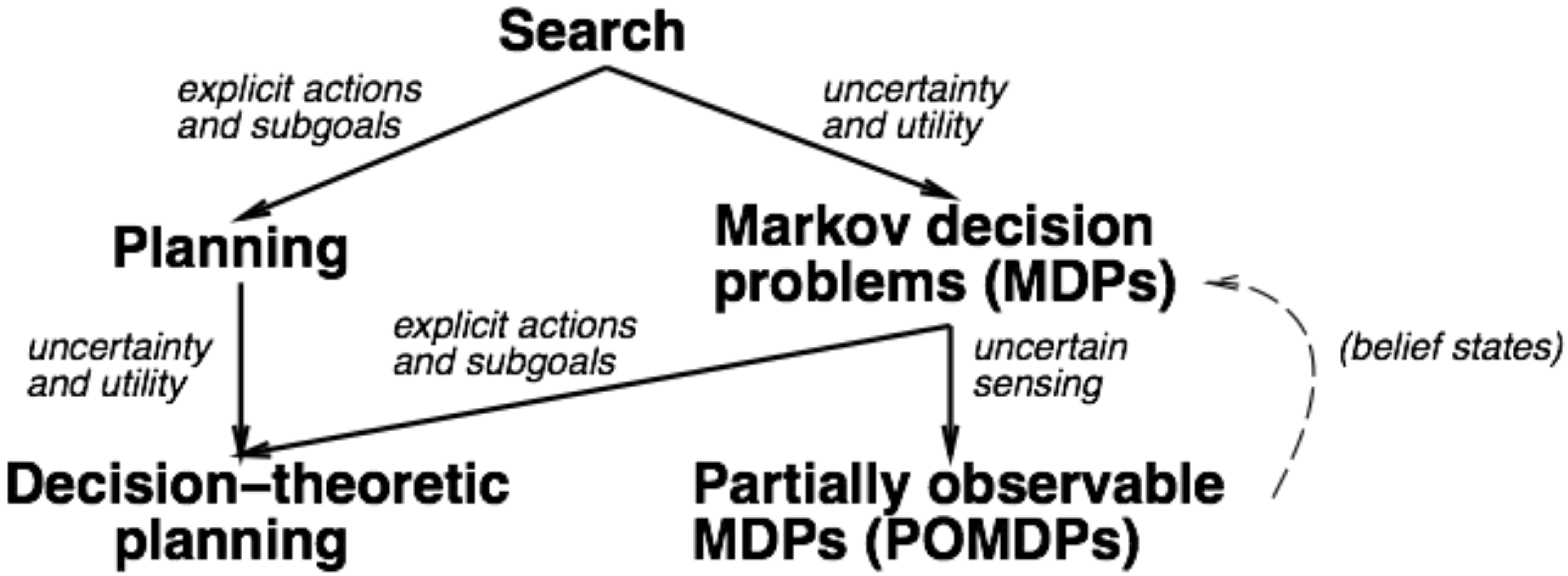
Previous session discussed problems with single decisions

Most interesting problems require the decision maker to make a series of decisions

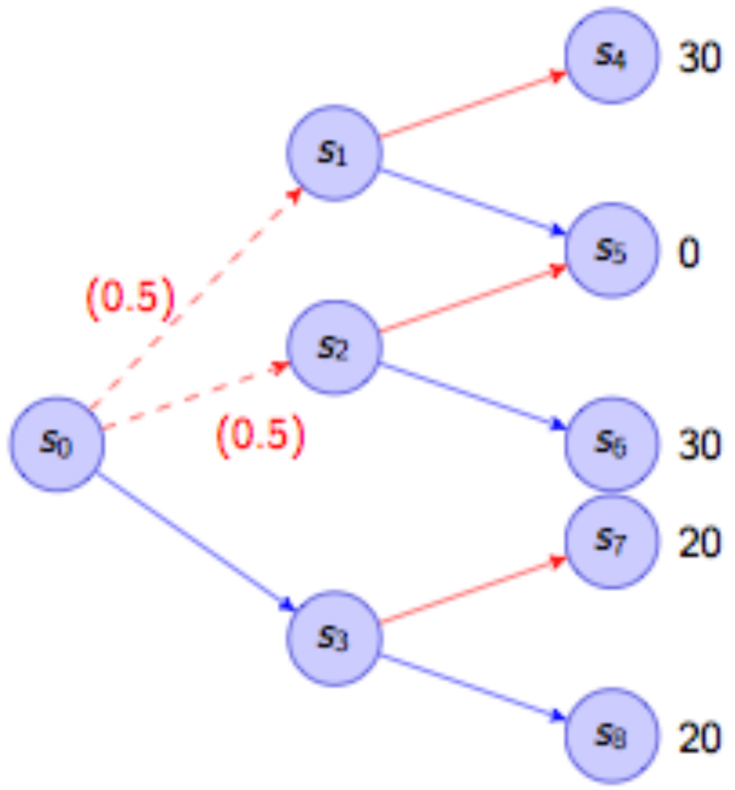
Same idea of maximum expected utility still holds, but requires reasoning about future sequences of actions and observations

This session will discuss *sequential decision problems* in *stochastic* environments

Sequential decision making


Search
explicit actions and subgoals
uncertainty and utility
Planning
Markov decision problems (MDPs)
uncertainty and utility
explicit actions and subgoals
uncertain sensing
(belief states)
Decision–theoretic planning
Partially observable MDPs (POMDPs)

Closed and open-loop planning



Closed loop: accounts for future state information (MDP)

Open loop: does not account for future state information (path planning)

Open loop plans do not always result in optimal behavior

U(r,r)=15

U(r,b)=15

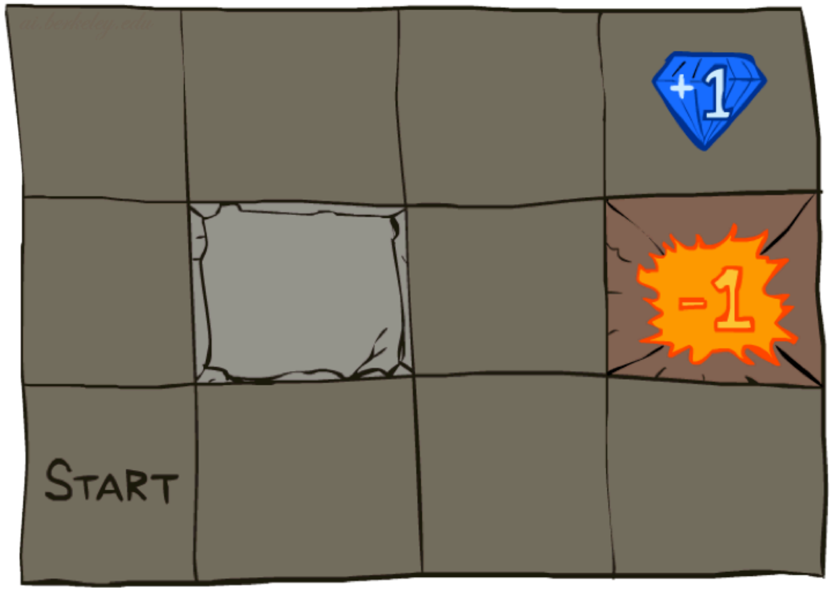
U(b,r)=20

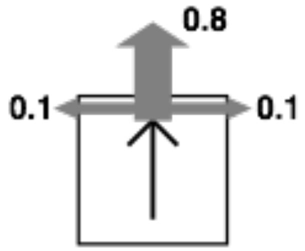
U(b,b)=20

MDP solution can increase utility to 30

Example: stochastic grid world

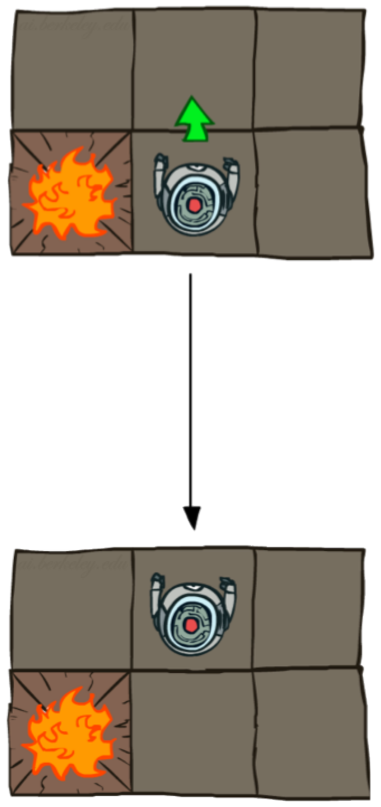
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Reward function can be anything. For ex:
    - Small "living" reward each step (can be negative)
    - Big rewards come at the end (good or bad)
- Goal: maximize (discounted) sum of rewards



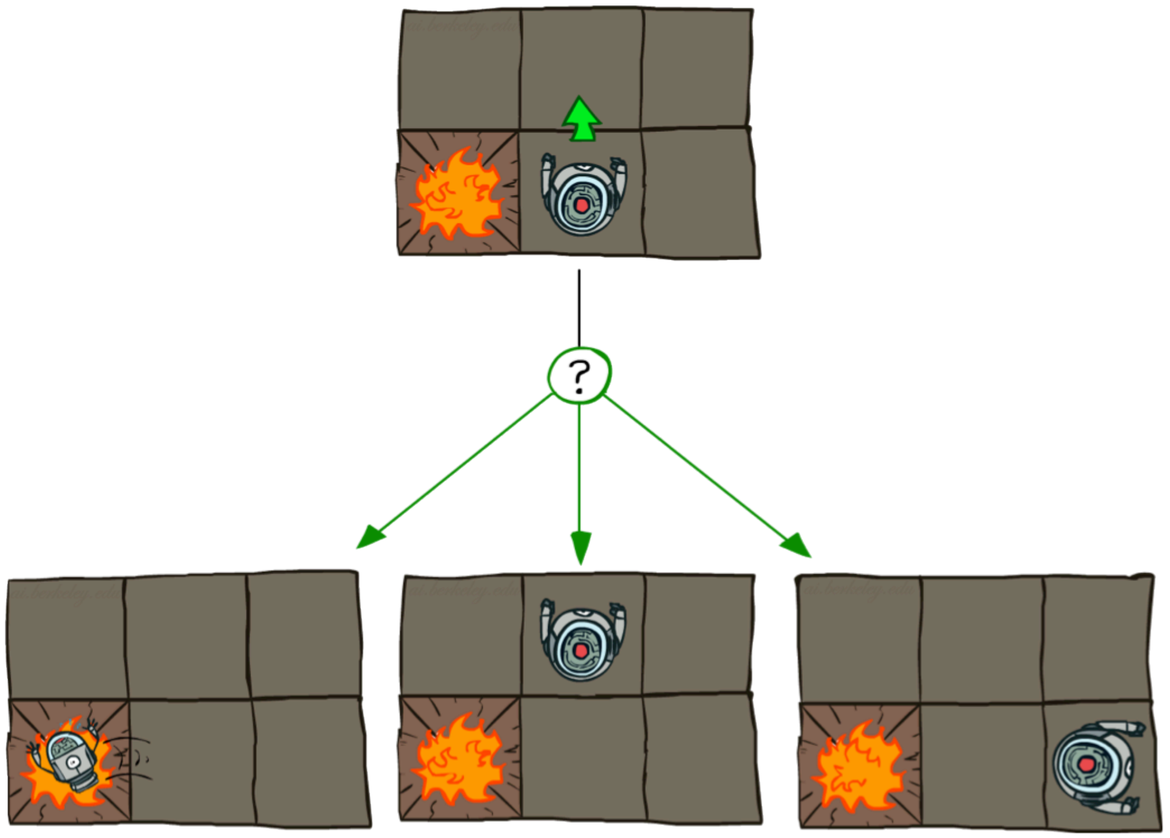


Stochastic actions

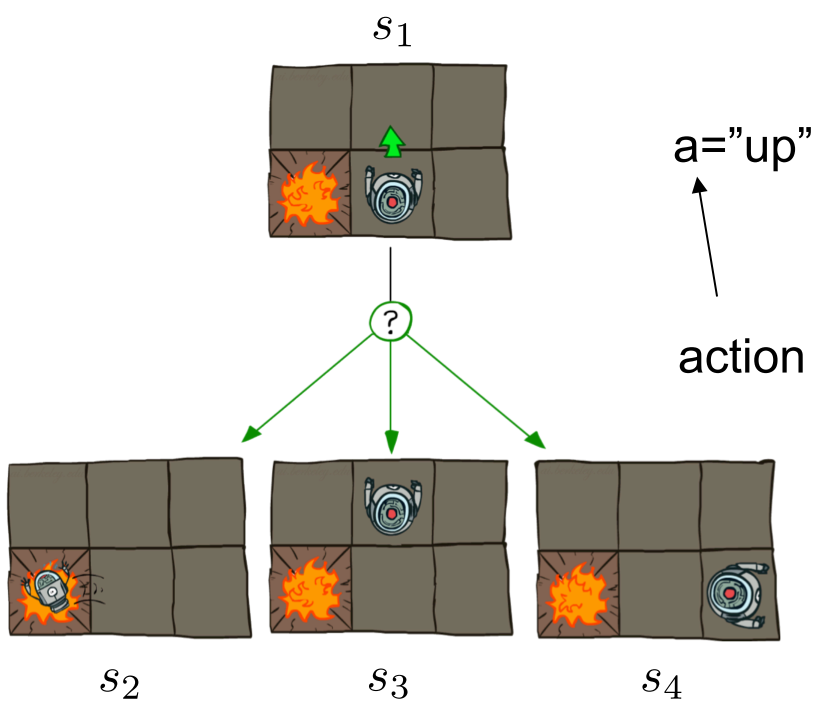
Deterministic Grid World



Stochastic Grid World



The transition function

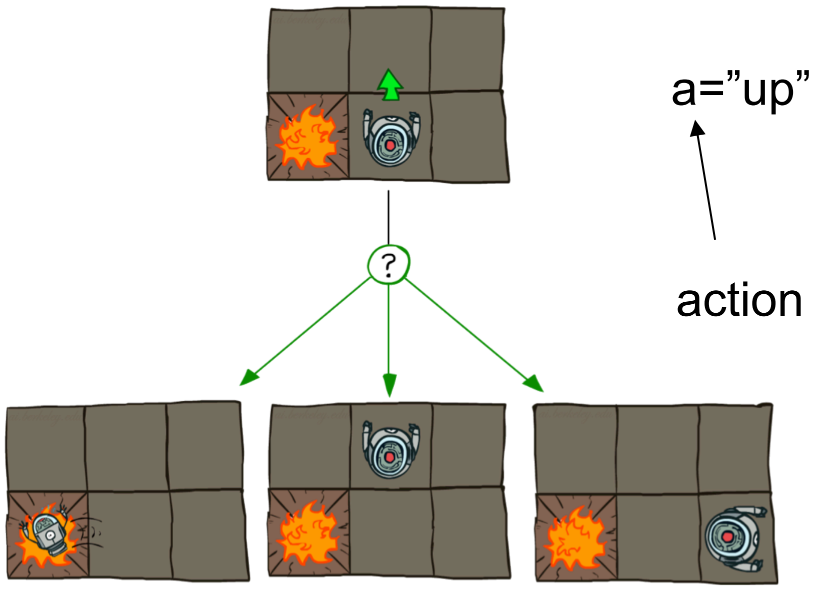


$$\begin{tikzcd}
 & s_1 \arrow[dl, "0.1"'] \arrow[d, "0.8"] \arrow[dr, "0.1"] & \\
s_2 & s_3 & s_4
\end{tikzcd}$$

Transition probabilities:

s'	$P(s' \mid s_1, a)$
s_2	0.1
s_3	0.8
s_4	0.1

The transition function



$$\begin{tikzcd} & s_1 \arrow[dl, "0.1"'] \arrow[d, "0.8"] \arrow[dr, "0.1"] & \\ s_2 & s_3 & s_4 \end{tikzcd}$$

Transition function: $T(s, a, s')$

– defines transition probabilities for each state,action pair

Transition probabilities:

s'	$P(s' \mid s_1, a)$
s_2	0.1
s_3	0.8
s_4	0.1

What is an *MDP*?

Technically, an MDP is a 4-tuple

An MDP (Markov Decision Process) defines a stochastic control problem: $M = (S, A, T, R)$

State set: $s \in S$

Action Set: $a \in A$

Transition function: $T : S \times A \times S \to \mathbb{R}_{\geq 0}$

Reward function: $R : S \times A \to \mathbb{R}_{\geq 0}$

Sometimes a start state and set of terminal states are given

What is an *MDP*?

Technically, an MDP is a 4-tuple

An MDP (Markov Decision Process) defines a stochastic control problem: $M = (S, A, T, R)$

State set: $s \in S$

Action Set: $a \in A$

Probability of going from *s* to *s'* when executing action *a*

Transition function: $T : S \times A \times S \rightarrow \mathbb{R}_{\geq 0}$

Reward function: $R : S \times A \rightarrow \mathbb{R}_{\geq 0}$

$$\sum_{s' \in S} T(s, a, s') = 1$$

What is an *MDP*?

Technically, an MDP is a 4-tuple

An MDP (Markov Decision Process) defines a stochastic control problem: $M = (S, A, T, R)$

State set: $s \in S$

Action Set: $a \in A$

Probability of going from *s* to *s′* when executing action *a*

Transition function: $T : S \times A \times S \rightarrow \mathbb{R}_{\geq 0}$

Reward function: $R : S \times A \rightarrow \mathbb{R}_{\geq 0}$

$$\sum_{s' \in S} T(s, a, s') = 1$$

But, what is the objective?

What is an *MDP*?

Technically, an MDP is a 4-tuple

An MDP (Markov Decision Process) defines a stochastic control problem: $M = (S, A, T, R)$

State set: $s \in S$

Action Set: $a \in A$

Probability of going from *s* to *s'* when executing action *a*

Transition function: $T : S \times A \times S \to \mathbb{R}_{\geq 0}$

Reward function: $R : S \times A \to \mathbb{R}_{\geq 0}$

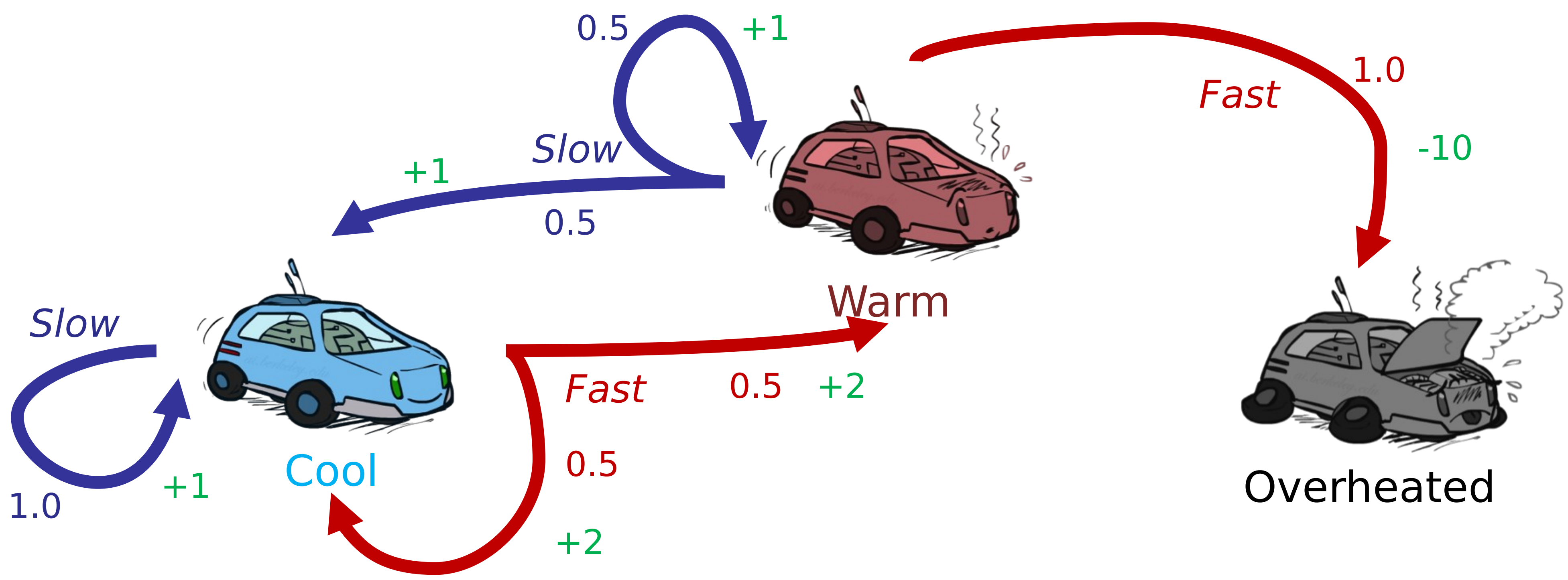
$$\sum_{s' \in S} T(s, a, s') = 1$$

<u>Objective</u>: calculate a strategy for acting so as to maximize the (discounted) sum of future rewards.

– we will calculate a *policy* that will tell us how to act

Example

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



Another example

Own aircraft must choose to stay level, climb, or descend

At each step, *-1* for collision, *-0.01* for climb or descend, *0* for staying level

State determined by altitude, closure rate, and vertical rates

Intruder aircraft flies around randomly

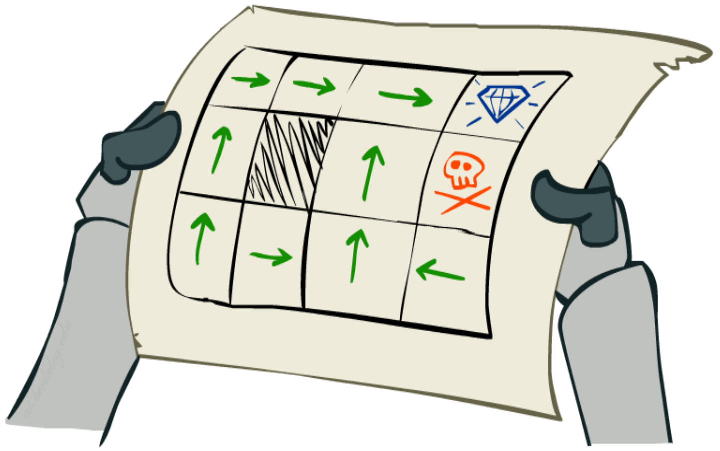
Optimal behavior determined by reward and transition model

What is a *policy*?

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

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

- Expectimax didn't compute entire policies
  - It computed the action for a single state only



This policy is optimal when R(s, a, s') = -0.03 for all non-terminal states

Why is it Markov?

- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means action outcomes depend only on the current state

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

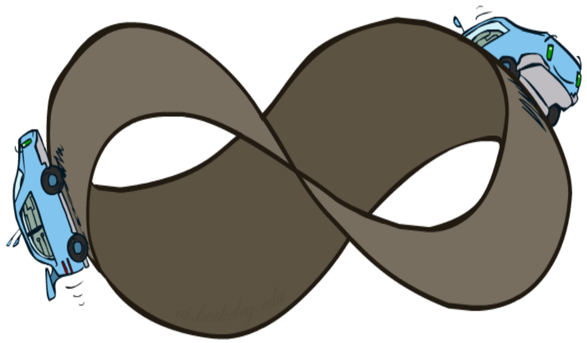
$$= \quad P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

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

Andrey Markov
(1856-1922)

Infinite utilities

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

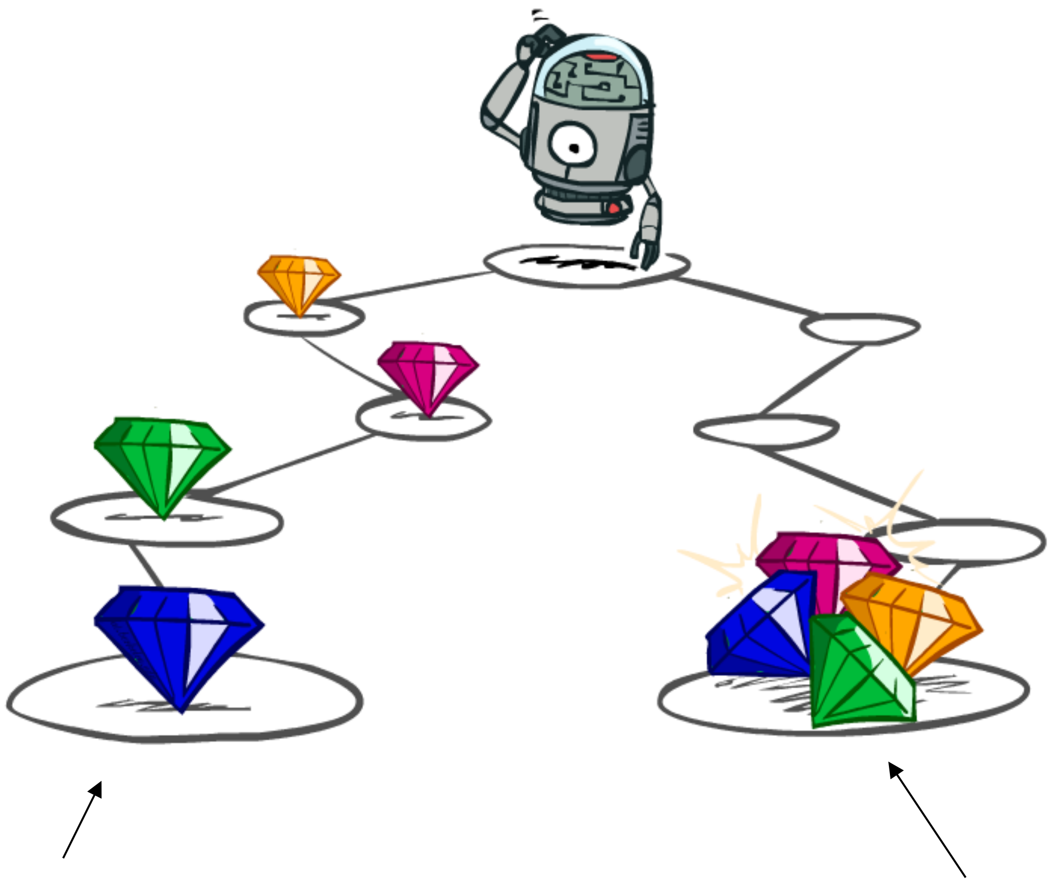


- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)
  - Discounting: use $0 < \gamma < 1$

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

    - Smaller γ means smaller “horizon” – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

Discounting rewards



Is this better?

Or is this better?

In general: how should we balance amount of reward vs how soon it is obtained?

Discounting rewards

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

1

Worth Now

γ

Worth Next Step

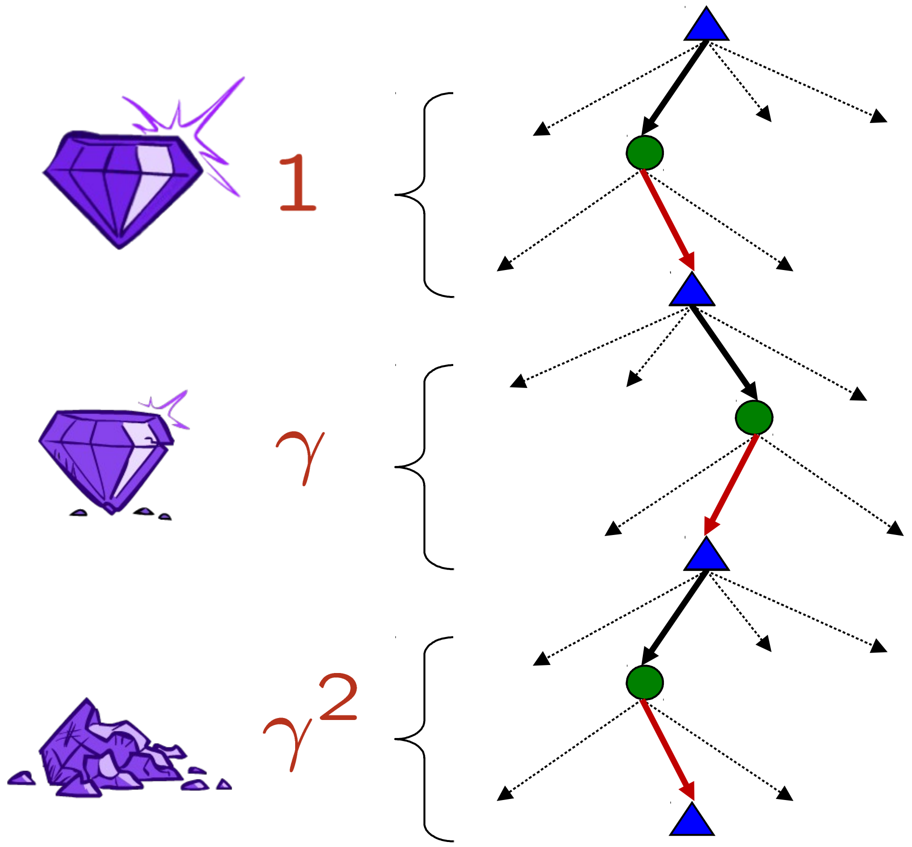
γ^2

Worth In Two Steps

Where, for example: $\gamma \approx 0.9$

Discounting rewards

- 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 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



Discounting rewards

In general:

$$\underbrace{U_t}_{\text{Utility}} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$$

$$= \sum_{k=0}^{\infty} \gamma^k r_{t+1+k}$$

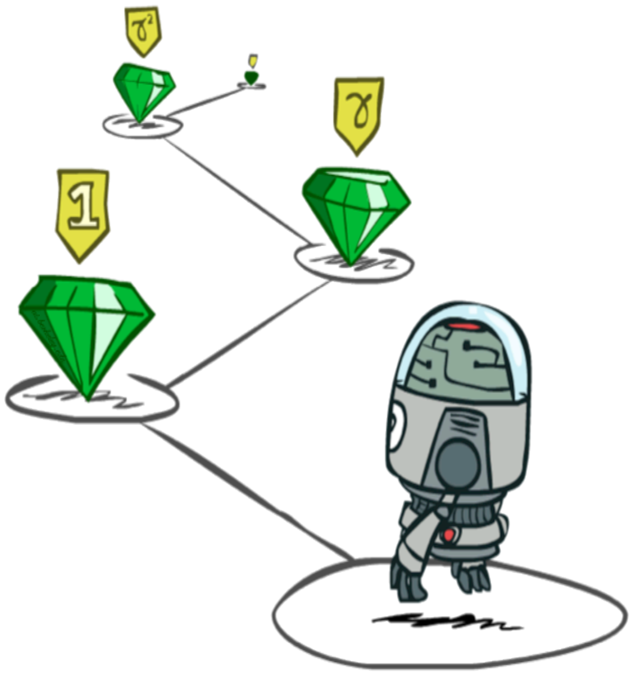
Stationary preferences

Theorem: if we assume stationary preferences:

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$

$$\Updownarrow$$

$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$



Then: there are only two ways to define utilities

Additive utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$

Discounted utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots$

Models of optimal behavior

In the *finite-horizon* model, agent should optimize expected reward for the next *H* steps: $\mathbb{E}\left(\sum_{t=0}^{H} r_t\right)$

- Continuously executing *H*-step optimal actions is known as receding horizon control

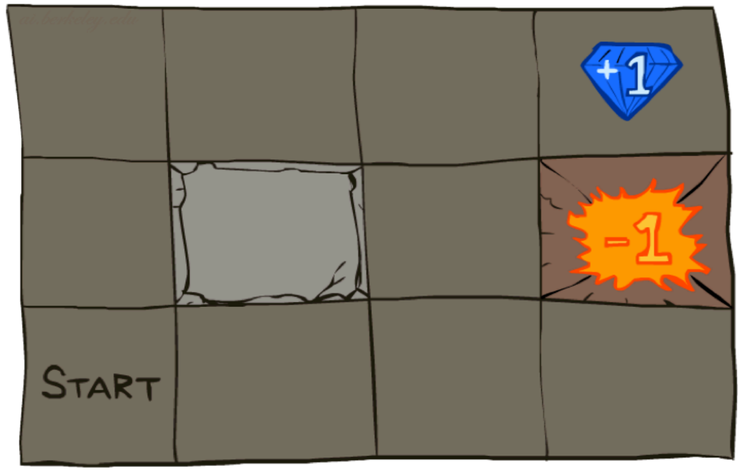
In the *infinite-horizon* discounted model agent should optimize: $\mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$

- Discount factor $0 \leq \gamma < 1$ can be thought of as an interest rate (reward now is worth more than reward in the future)

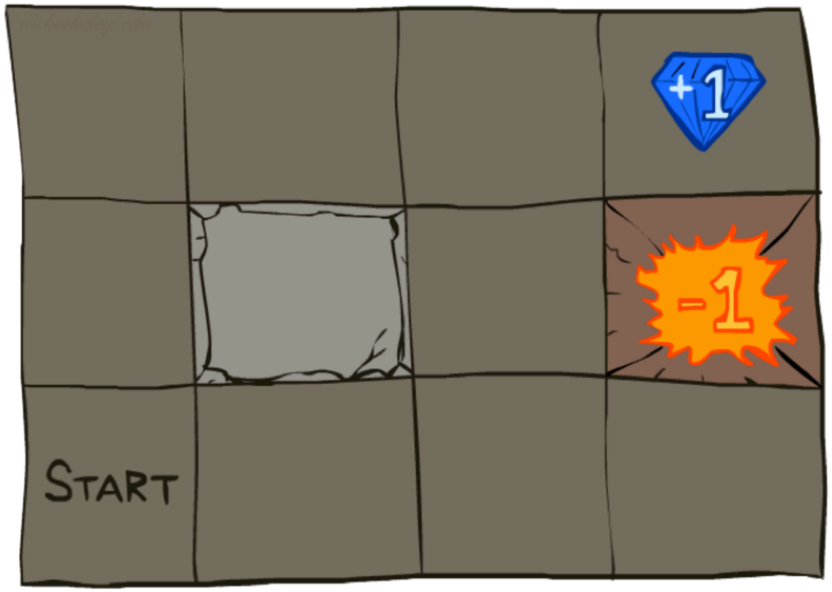
Choosing a reward function

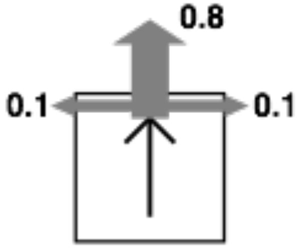
A few possibilities:
– all reward on goal/firepit
– negative reward everywhere
  except terminal states
– gradually increasing reward as you approach the goal

In general:
– reward can be whatever you want



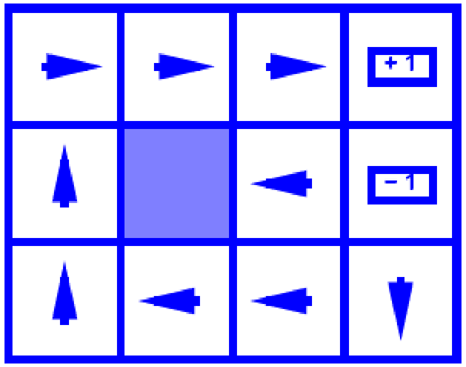
Examples of optimal policies



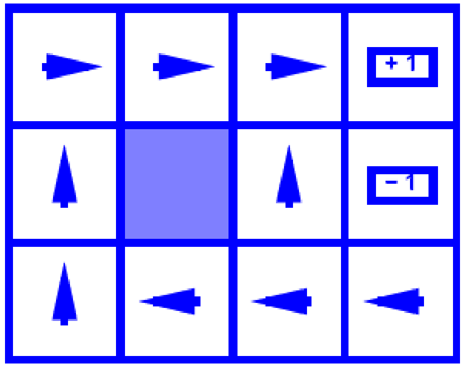


What happens if we change the “living” reward?

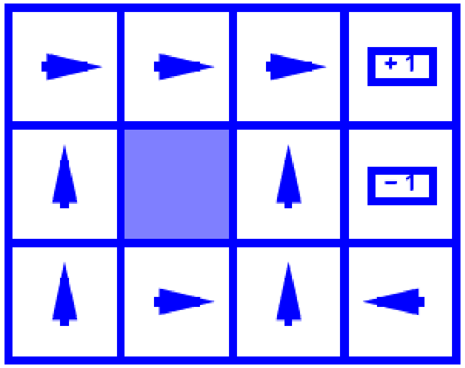
Examples of optimal policies



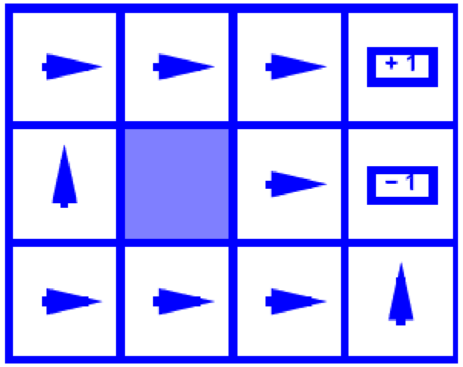
R(s) = -0.01



R(s) = -0.03



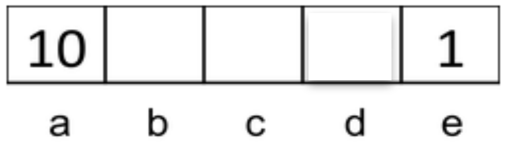
R(s) = -0.4



R(s) = -2.0

Discounting example

- 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?

Solving MDPs

- The value (utility) of a state s:

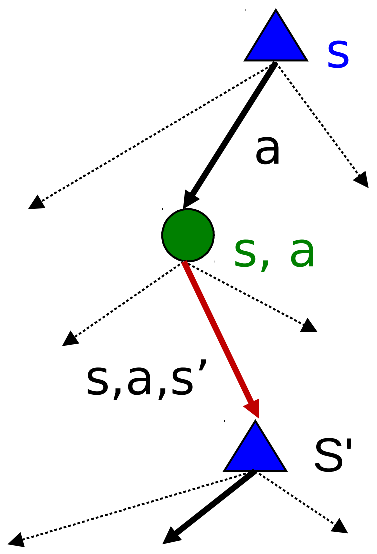
  $V^*(s)$ = expected utility starting in s and acting optimally

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

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

- The optimal policy:

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

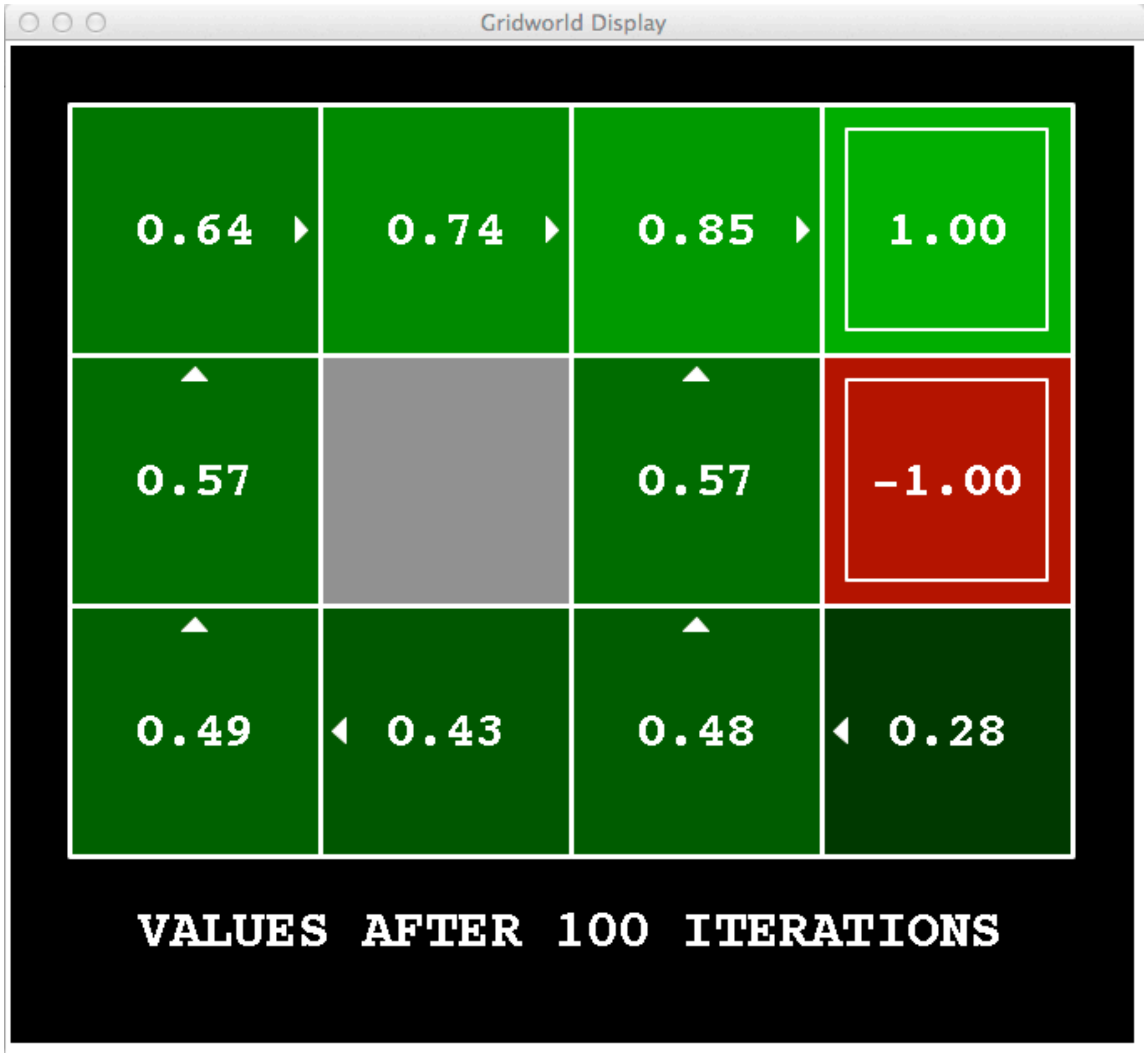


s is a *state*

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

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

Snapshot of Demo – Gridworld V Values

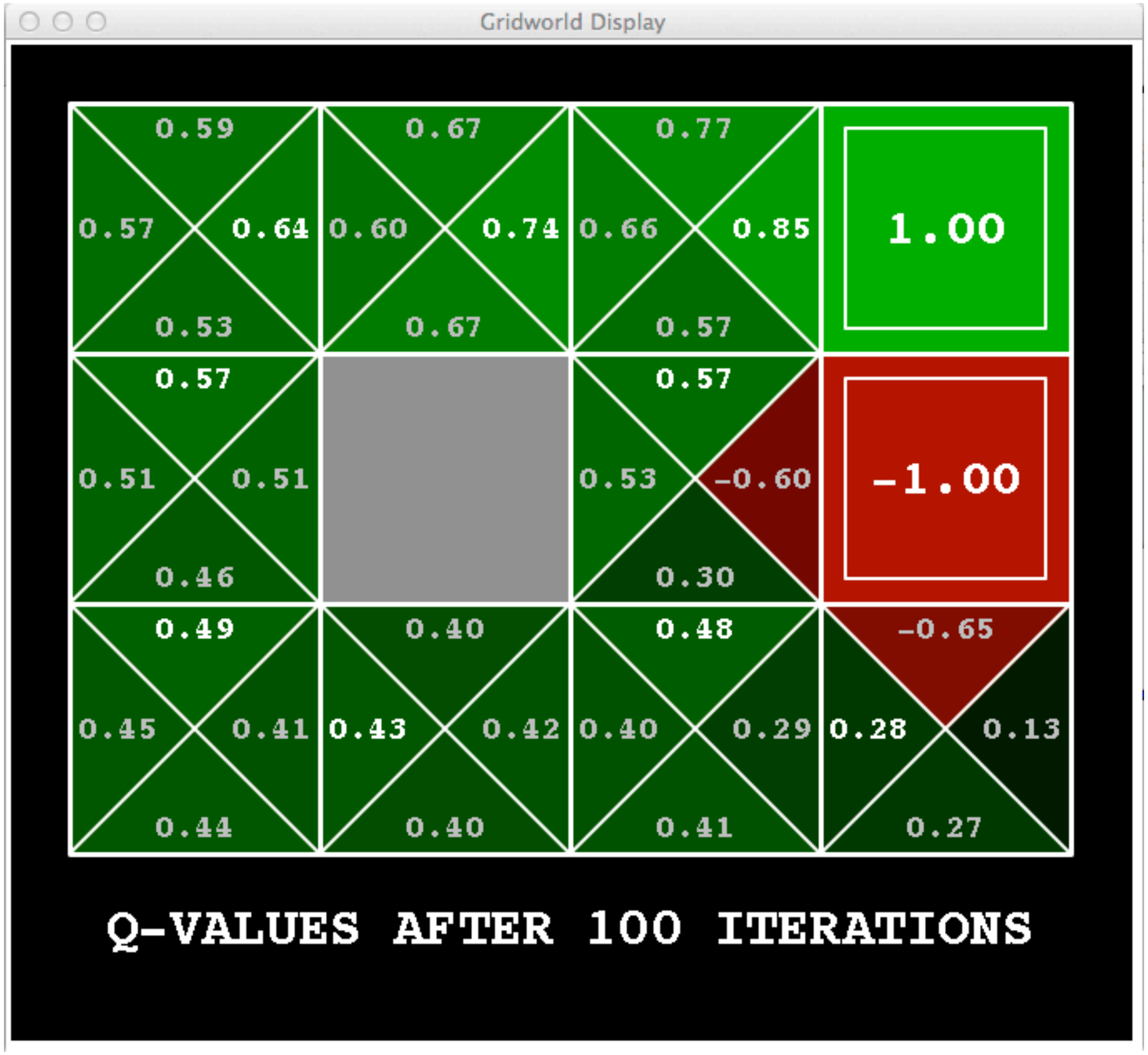


0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.57		▲ 0.57	-1.00
▲ 0.49	◀ 0.43	▲ 0.48	◀ 0.28

VALUES AFTER 100 ITERATIONS

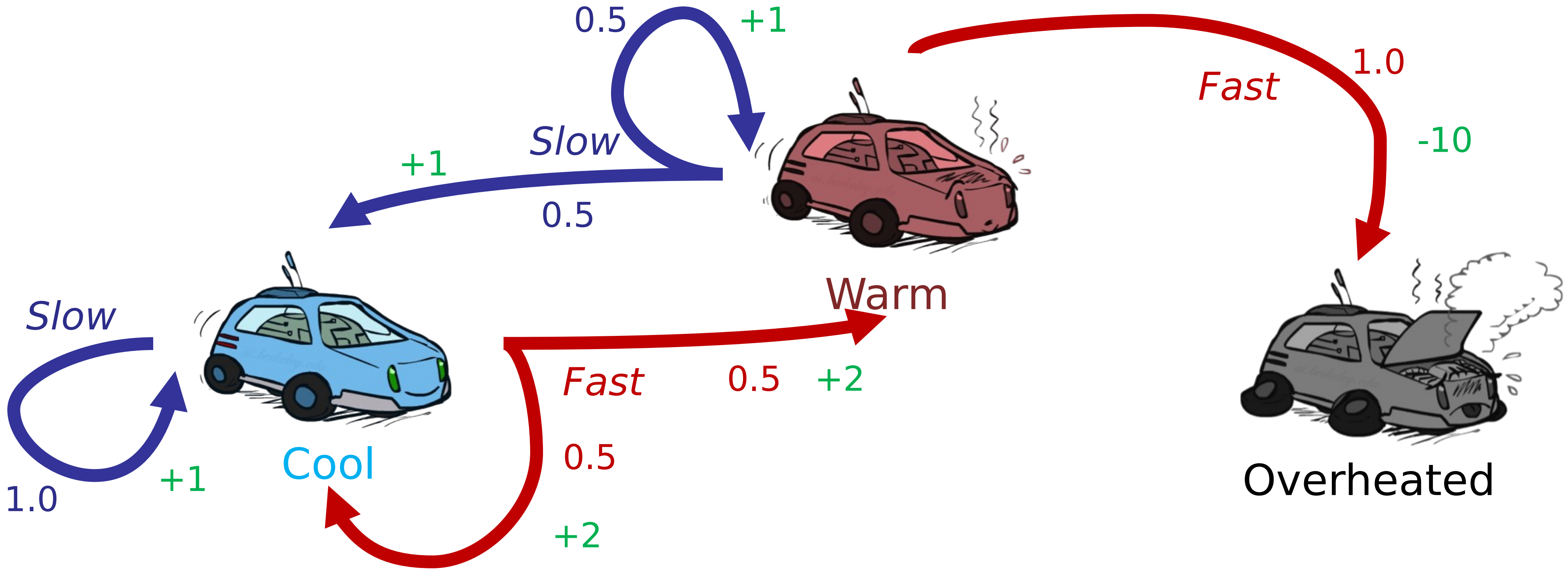
Noise = 0.2
Discount = 0.9
Living reward = 0

Snapshot of Demo – Gridworld V Values

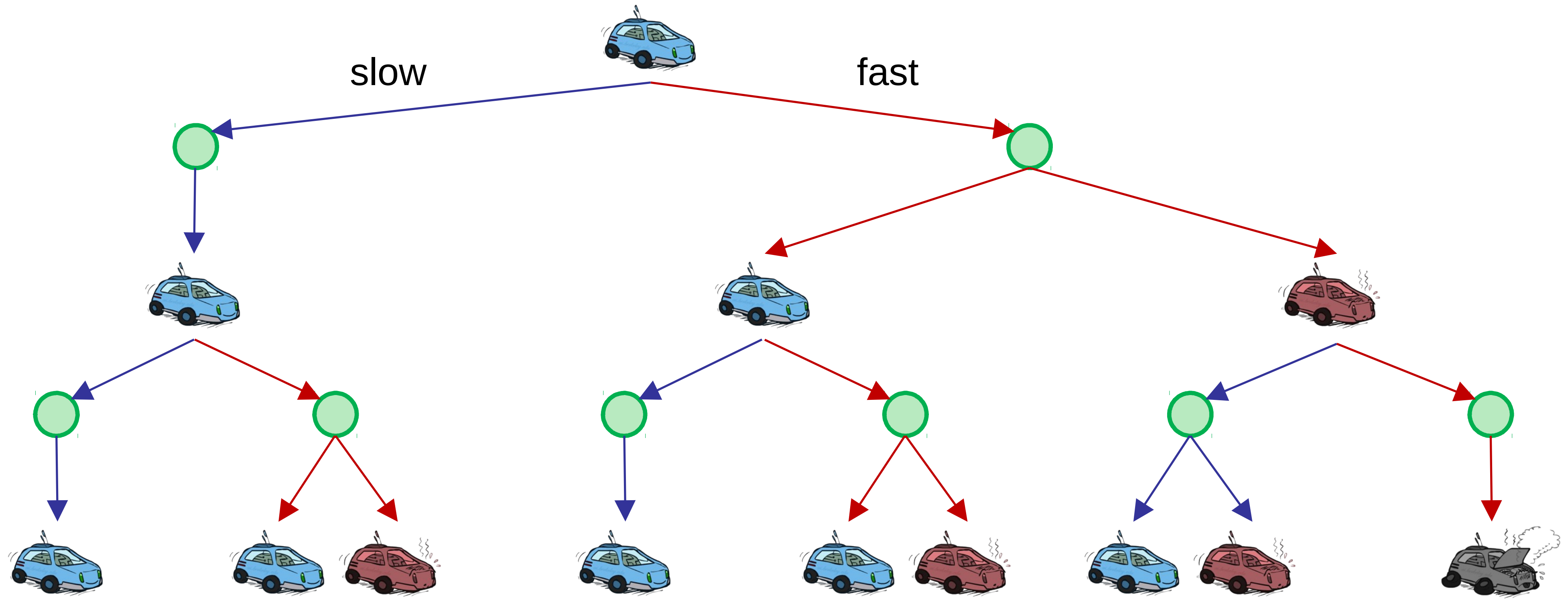


Noise = 0.2
Discount = 0.9
Living reward = 0

How would we solve this using expectimax?



How would we solve this using expectimax?



Problems w/ this approach?

How would we solve this using expectimax?

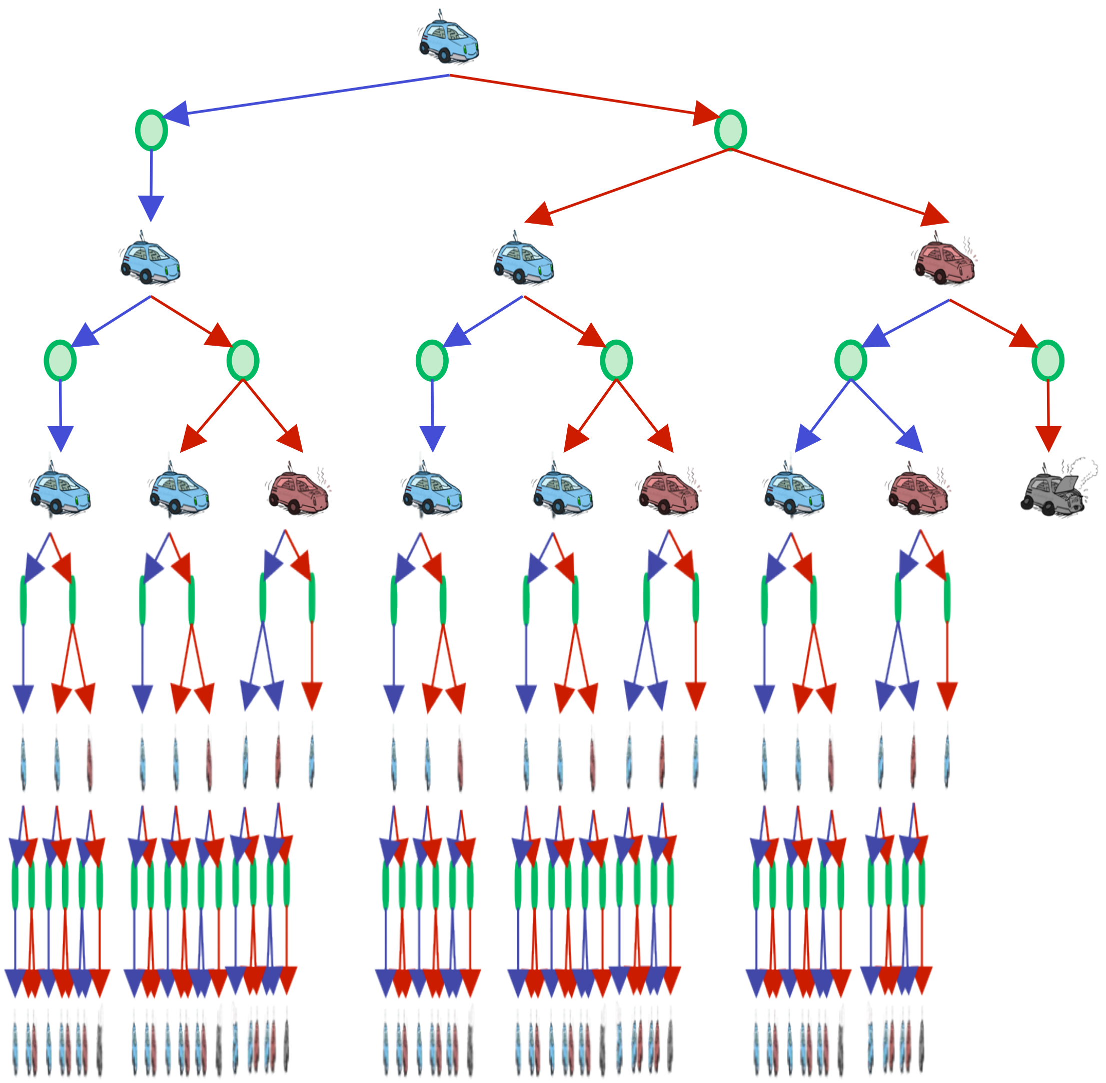
We're doing way too much work with expectimax!

Problem: States are repeated

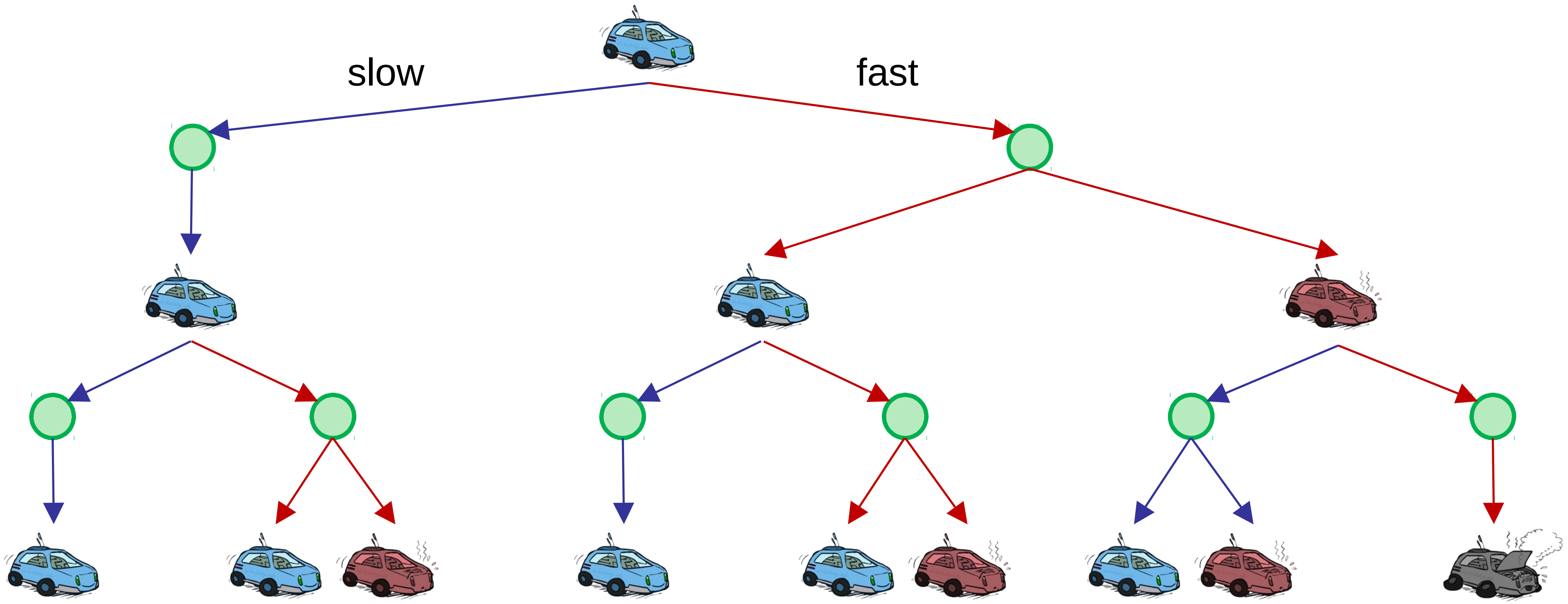
- Idea: Only compute needed quantities once

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 don't matter if $\gamma < 1$



How would we solve this using expectimax?

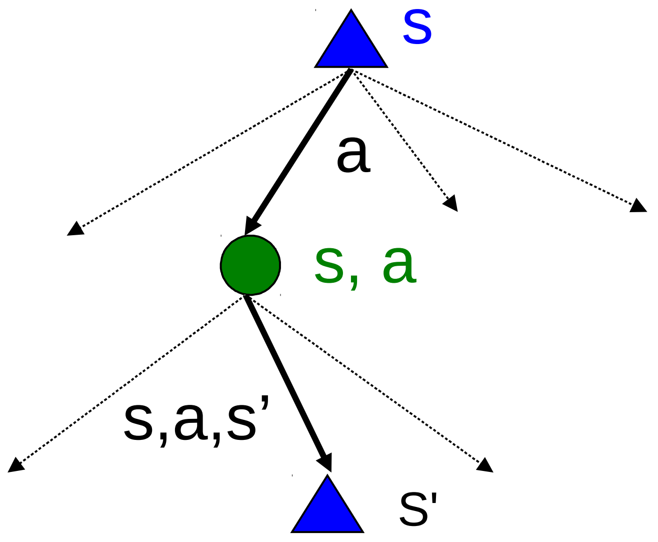


Problems w/ this approach:
– how deep do we search?
– how do we deal w/ loops?

Is there a better way?

Value iteration

We're going to calculate V* and/or Q* by repeatedly doing one-step expectimax.



Notice that the V* and Q* can be defined recursively:

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

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

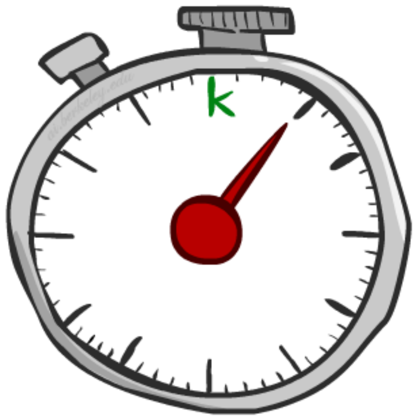
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

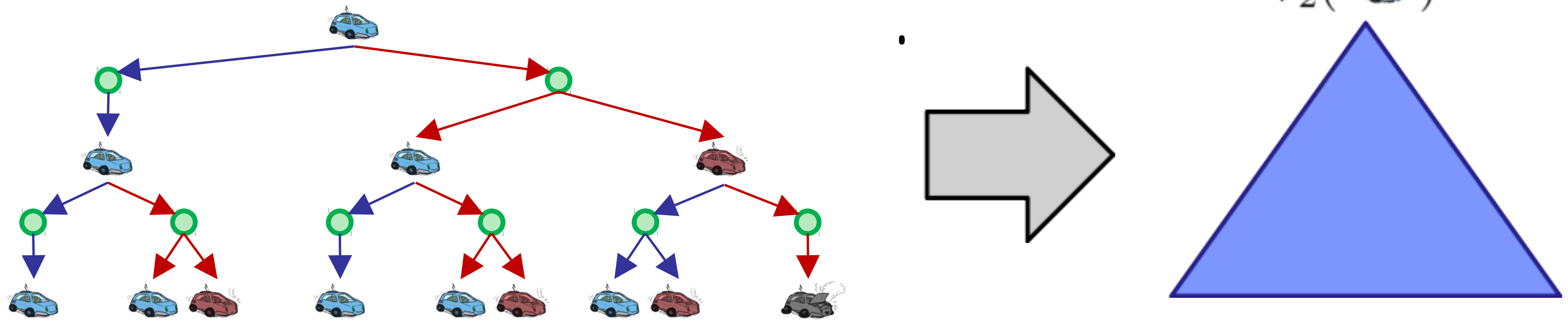
Called Bellman equations

– note that the above do not reference the optimal policy, π^*

Value iteration

- 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's what a depth-*k* expectimax would give from *s*





Value iteration

Value of *s* at *k* timesteps to go: $V_k(s)$

<u>Value iteration:</u>

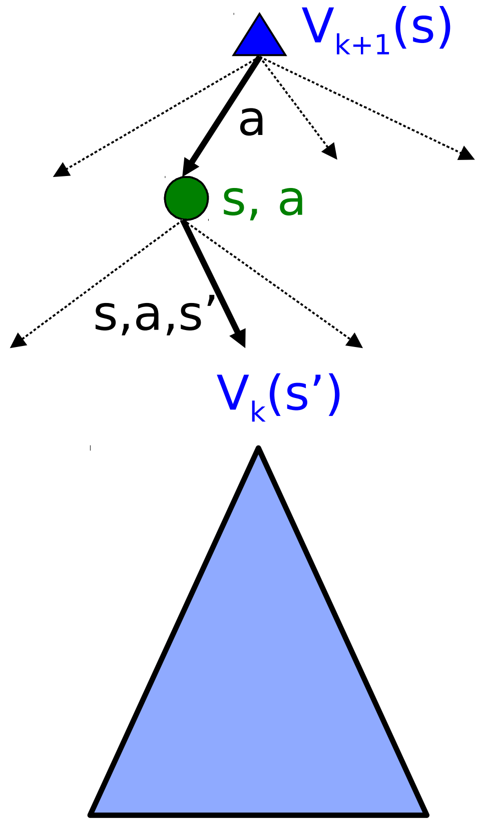
1. initialize $V_0(s) = 0$

2. $V_1(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_0(s') \right]$

3. $V_2(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_1(s') \right]$

4. ….

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



Value iteration

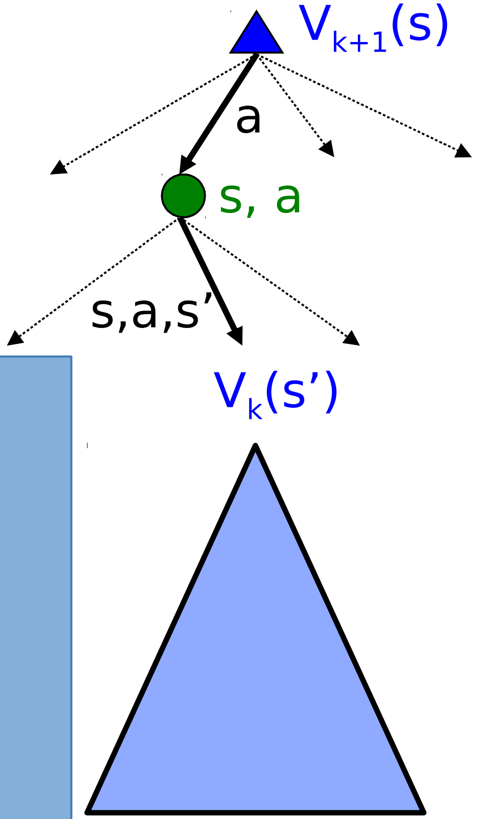
Value of *s* at *k* timesteps to go: $V_k(s)$

<u>Value iteration:</u>

1. initialize $V_0(s) = 0$
2. $V_1(s) \leftarrow$
3. $V_2(s) \leftarrow$
4. ....
5. $V_{k+1}(s)$

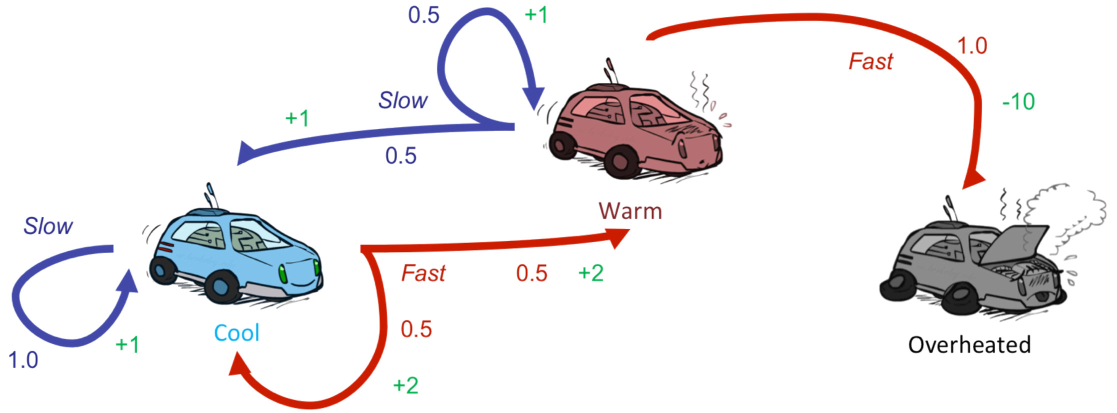
– This iteration converges! The value of each state converges to a unique optimal value.

– time complexity = $O(S^2A)$



Value iteration example

V_2			
V_1			
V_0	0	0	0

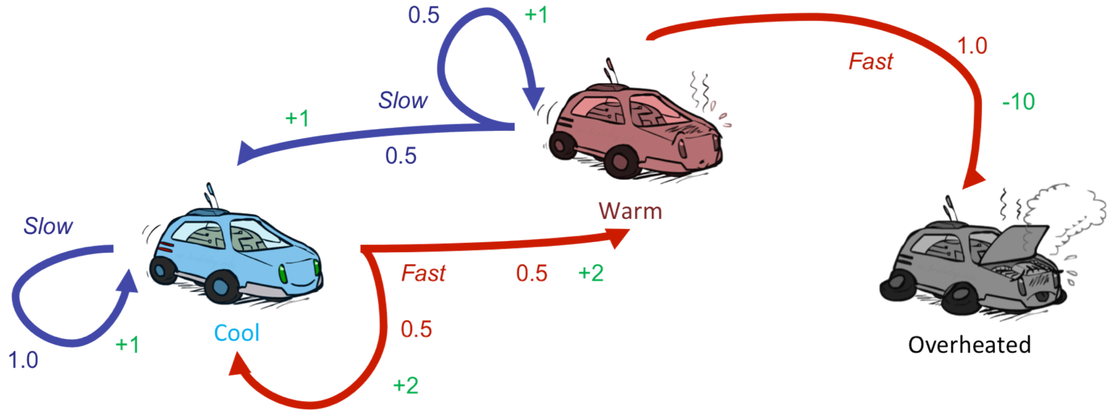


Assume no discount

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

Value iteration example

V_2			
V_1	2	1	0
V_0	0	0	0

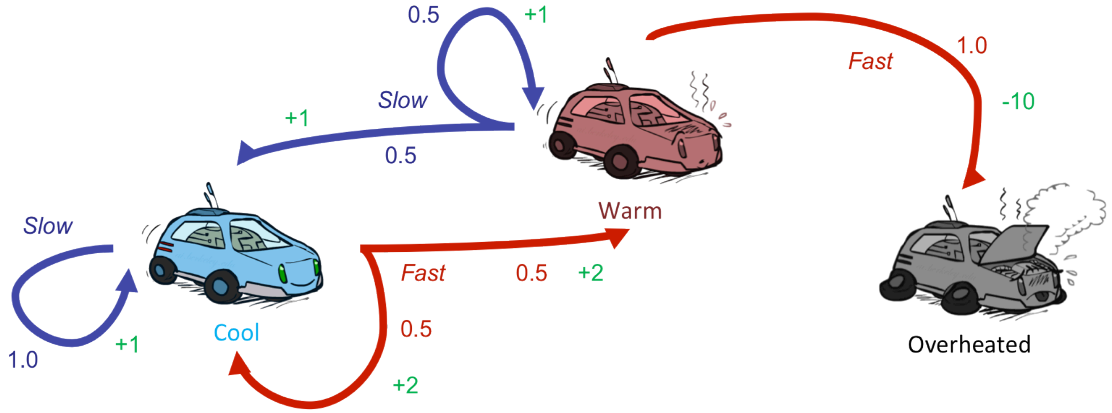


Assume no discount

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

Value iteration example

V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

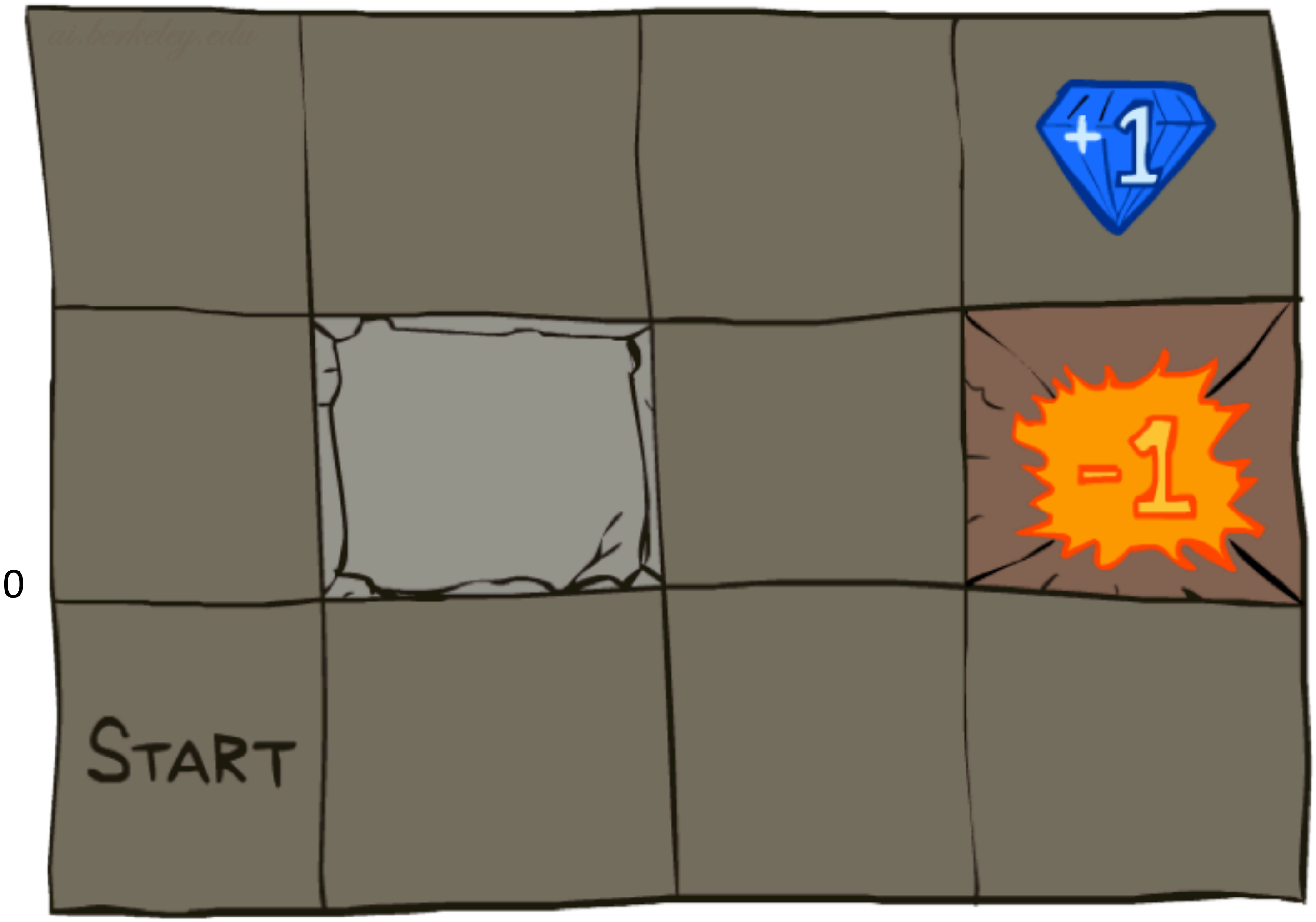


Assume no discount

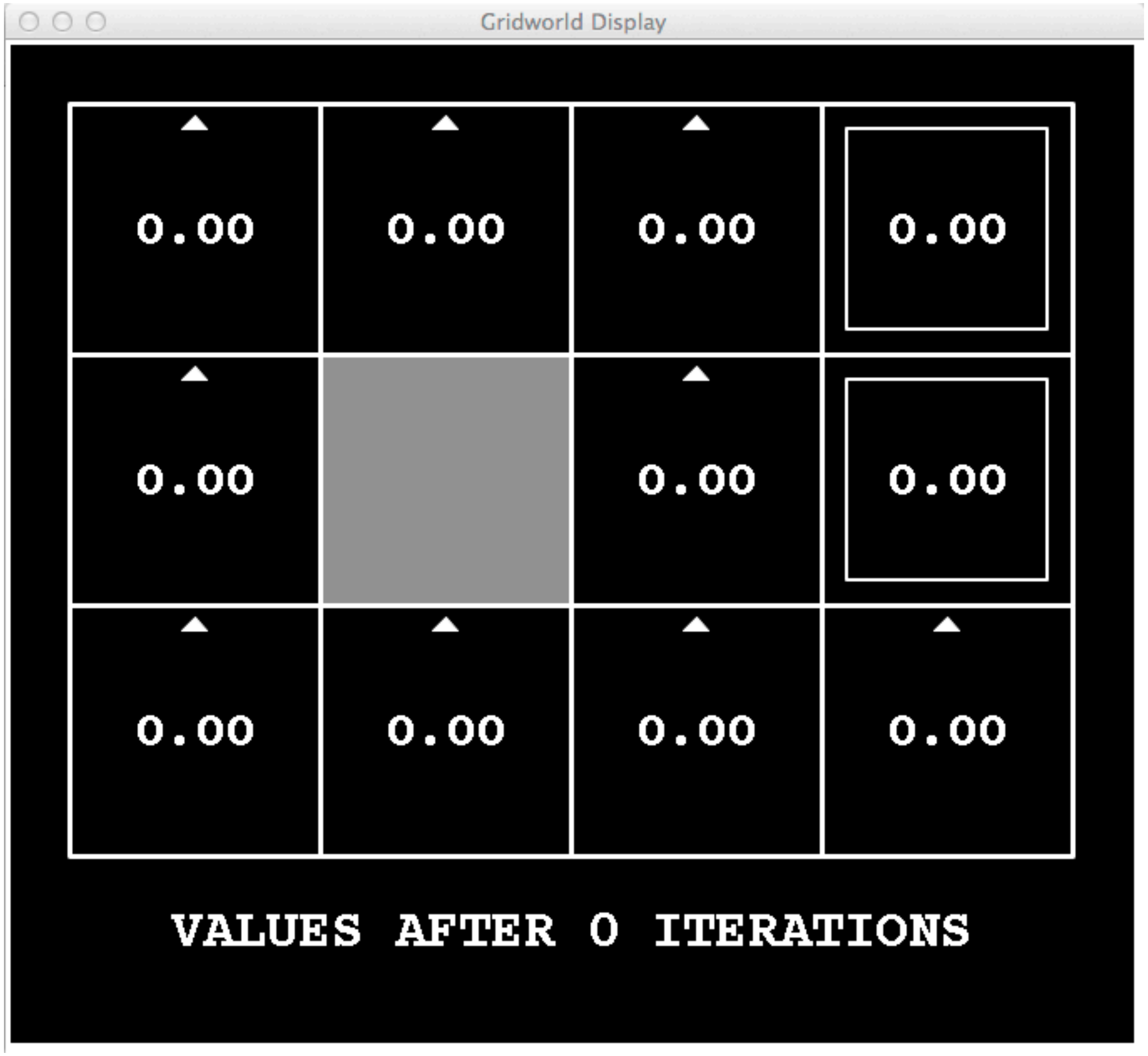
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

Back to the gridworld

Noise = 0.2
Discount = 0.9
Living reward = 0



Value iteration example

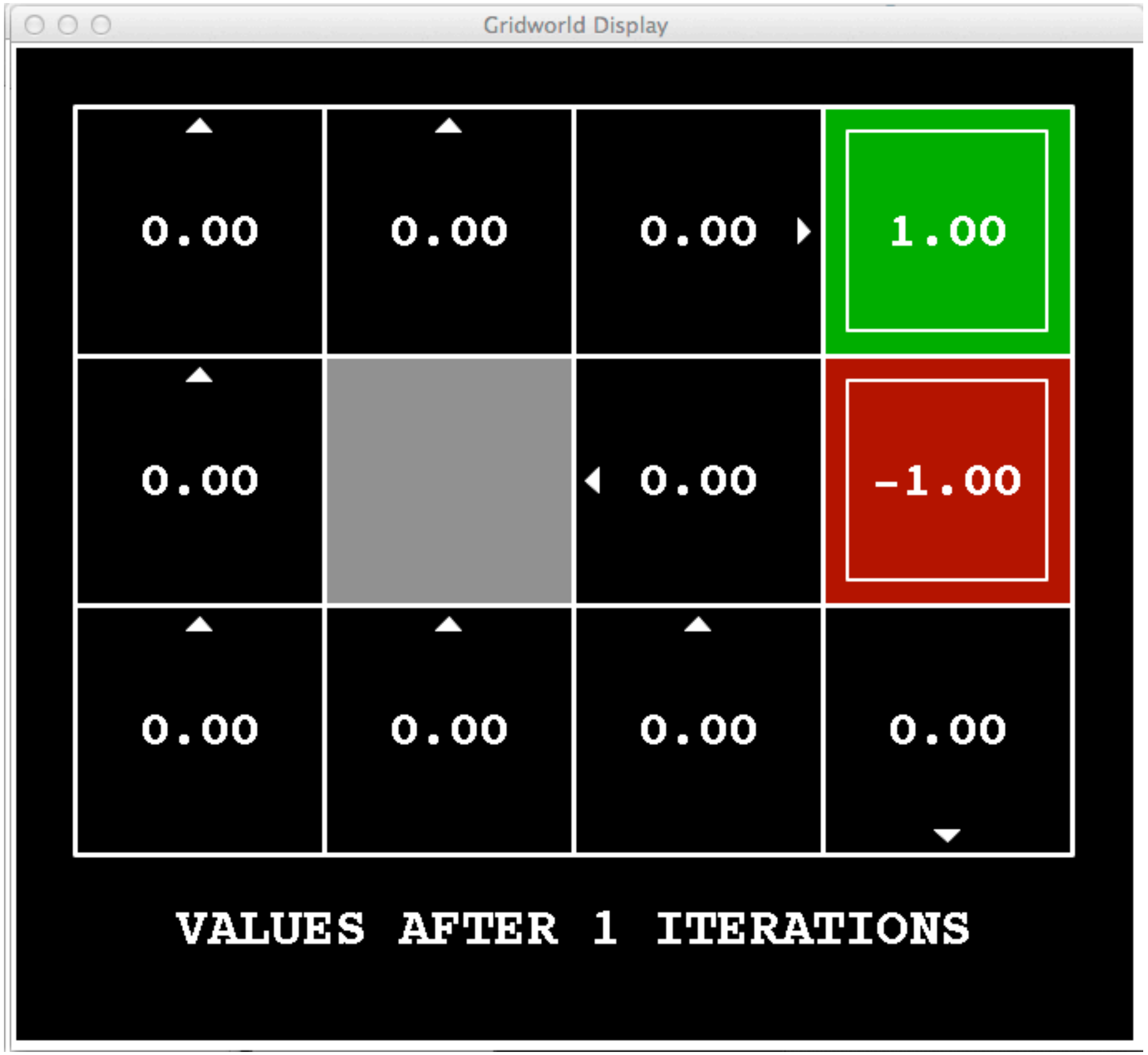


0.00	0.00	0.00	0.00
0.00		0.00	0.00
0.00	0.00	0.00	0.00

VALUES AFTER 0 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

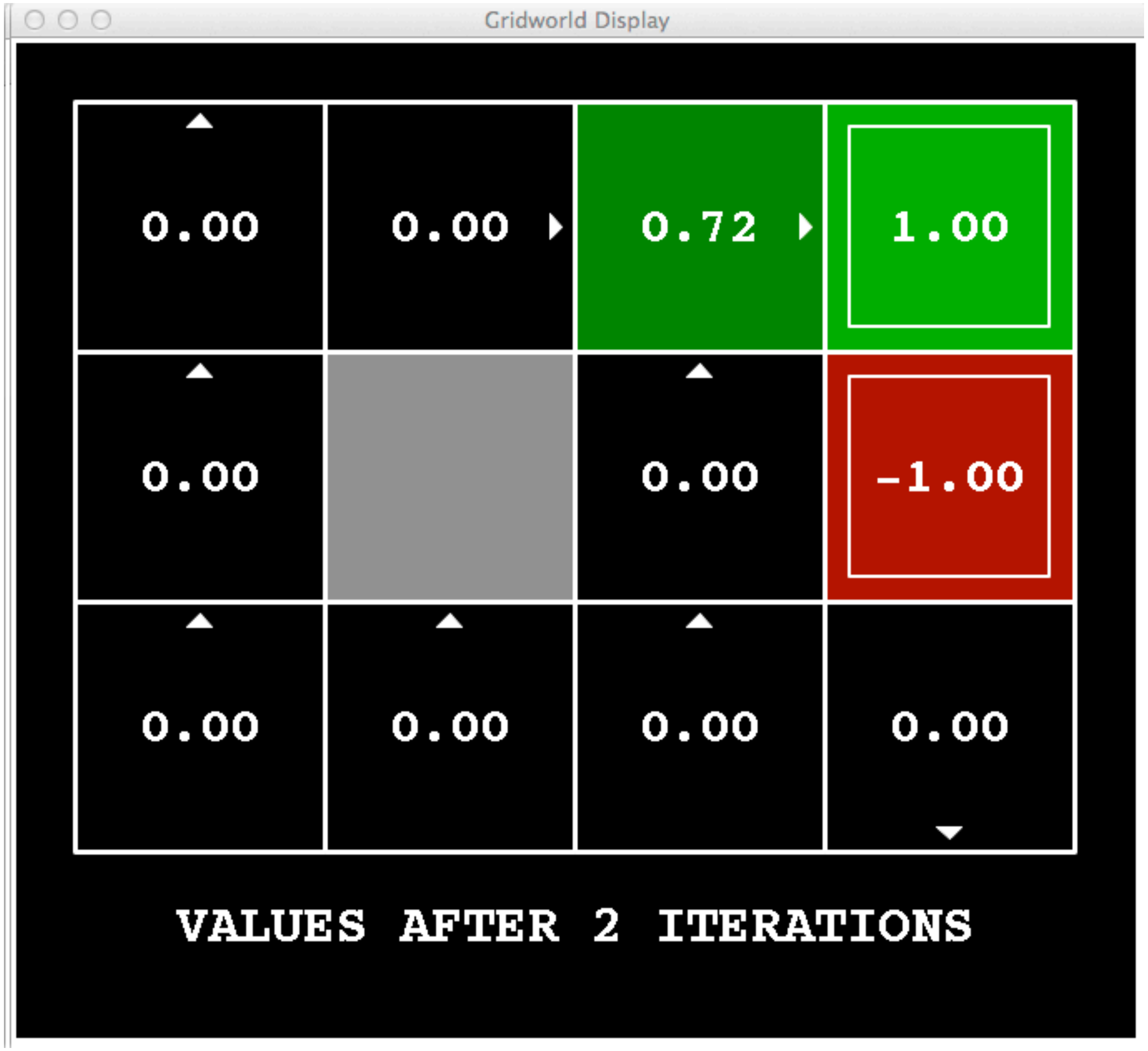
Value iteration example



0.00	0.00	0.00	1.00
0.00		0.00	-1.00
0.00	0.00	0.00	0.00

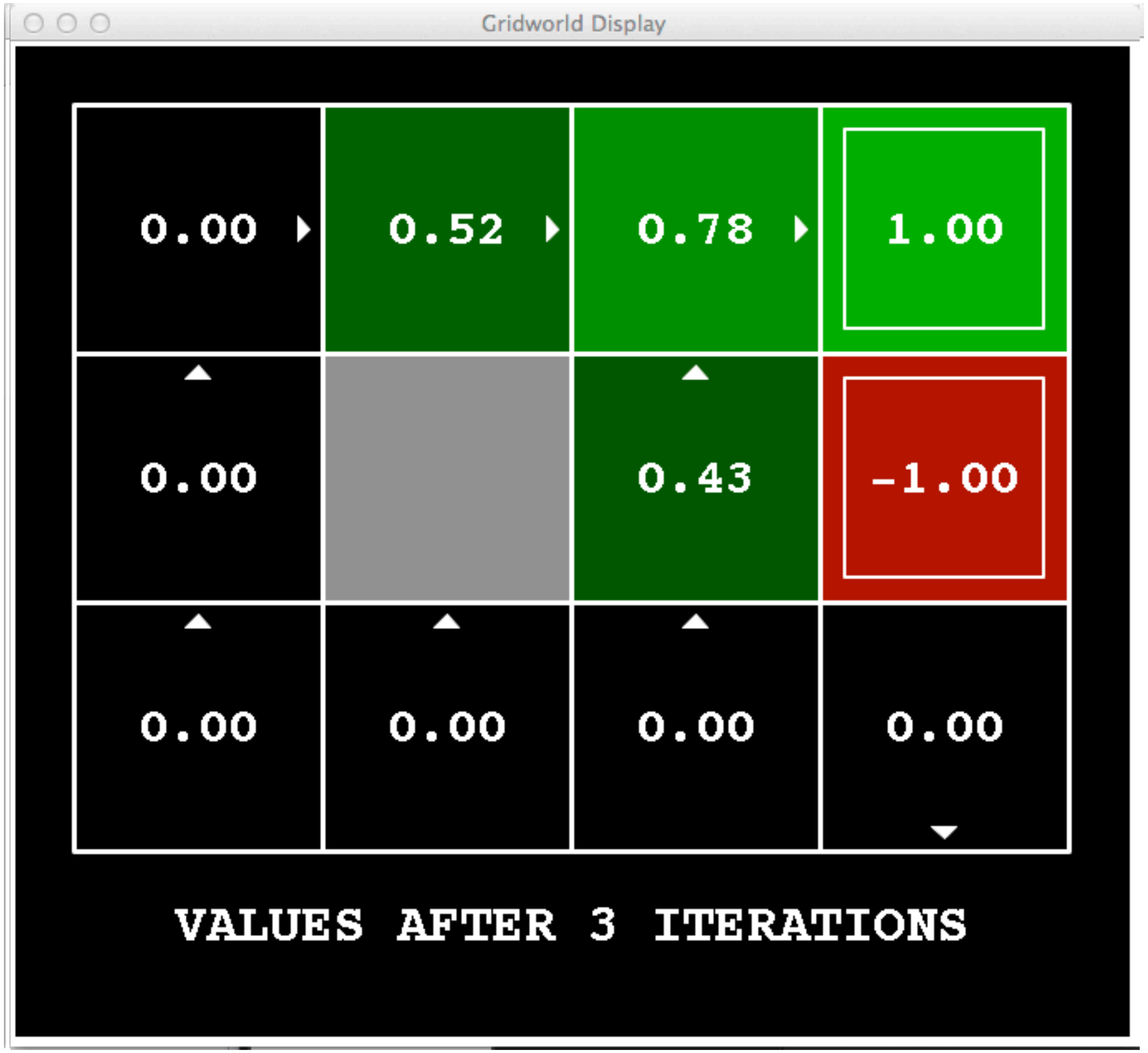
VALUES AFTER 1 ITERATIONS

Value iteration example

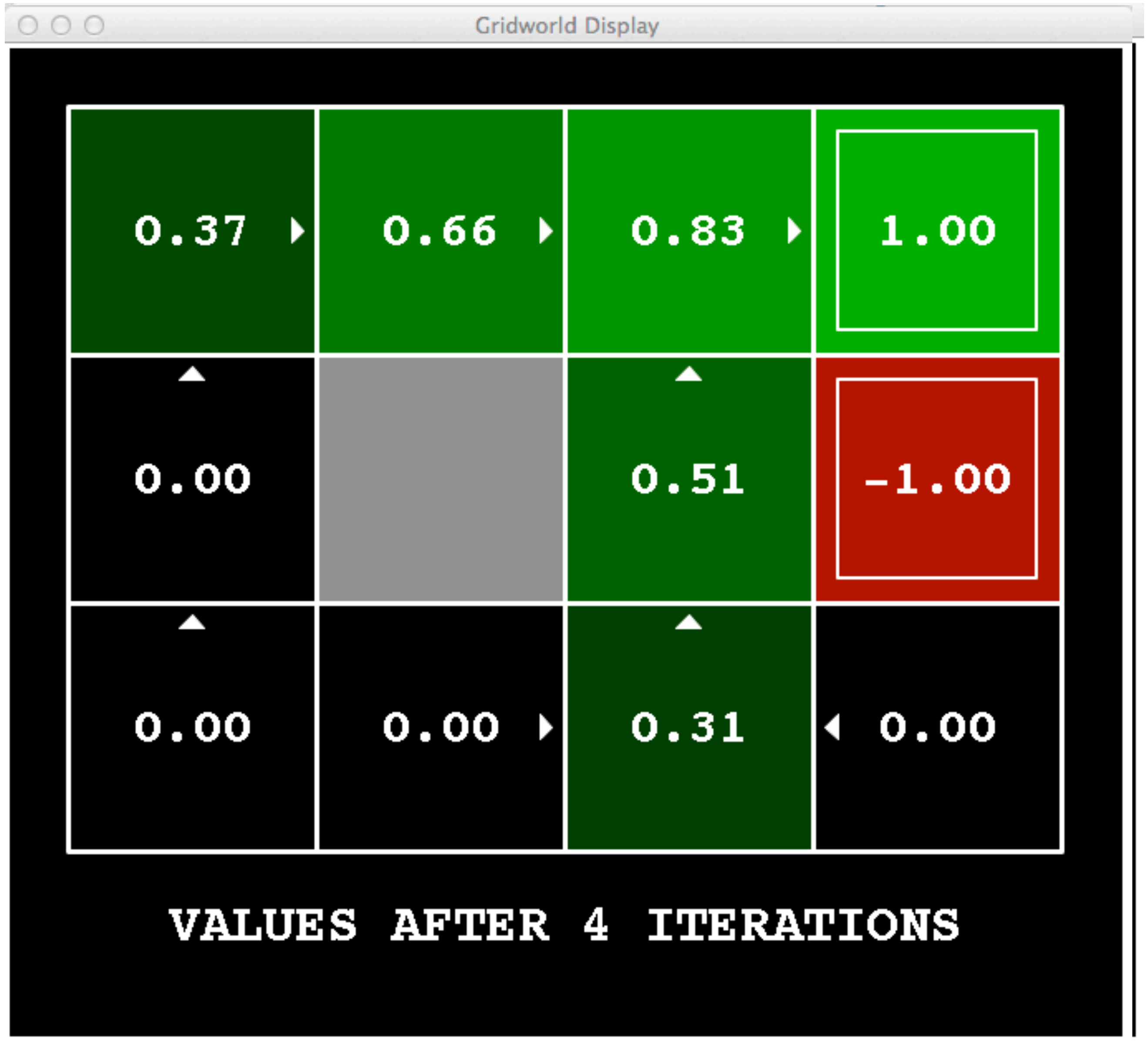


VALUES AFTER 2 ITERATIONS

Value iteration example



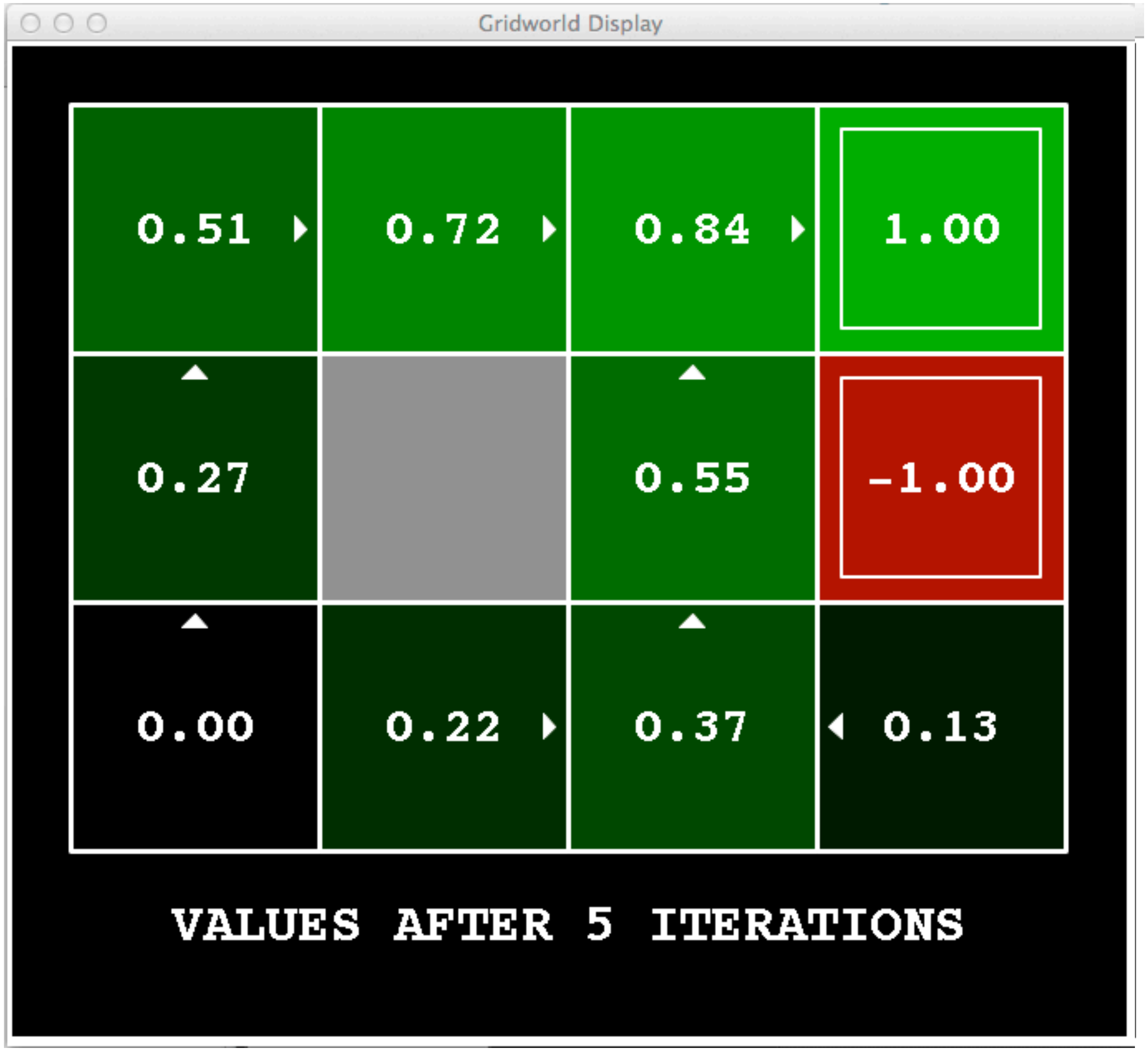
Value iteration example



0.37 ▶	0.66 ▶	0.83 ▶	1.00
▲ 0.00		▲ 0.51	-1.00
▲ 0.00	0.00 ▶	▲ 0.31	◀ 0.00

VALUES AFTER 4 ITERATIONS

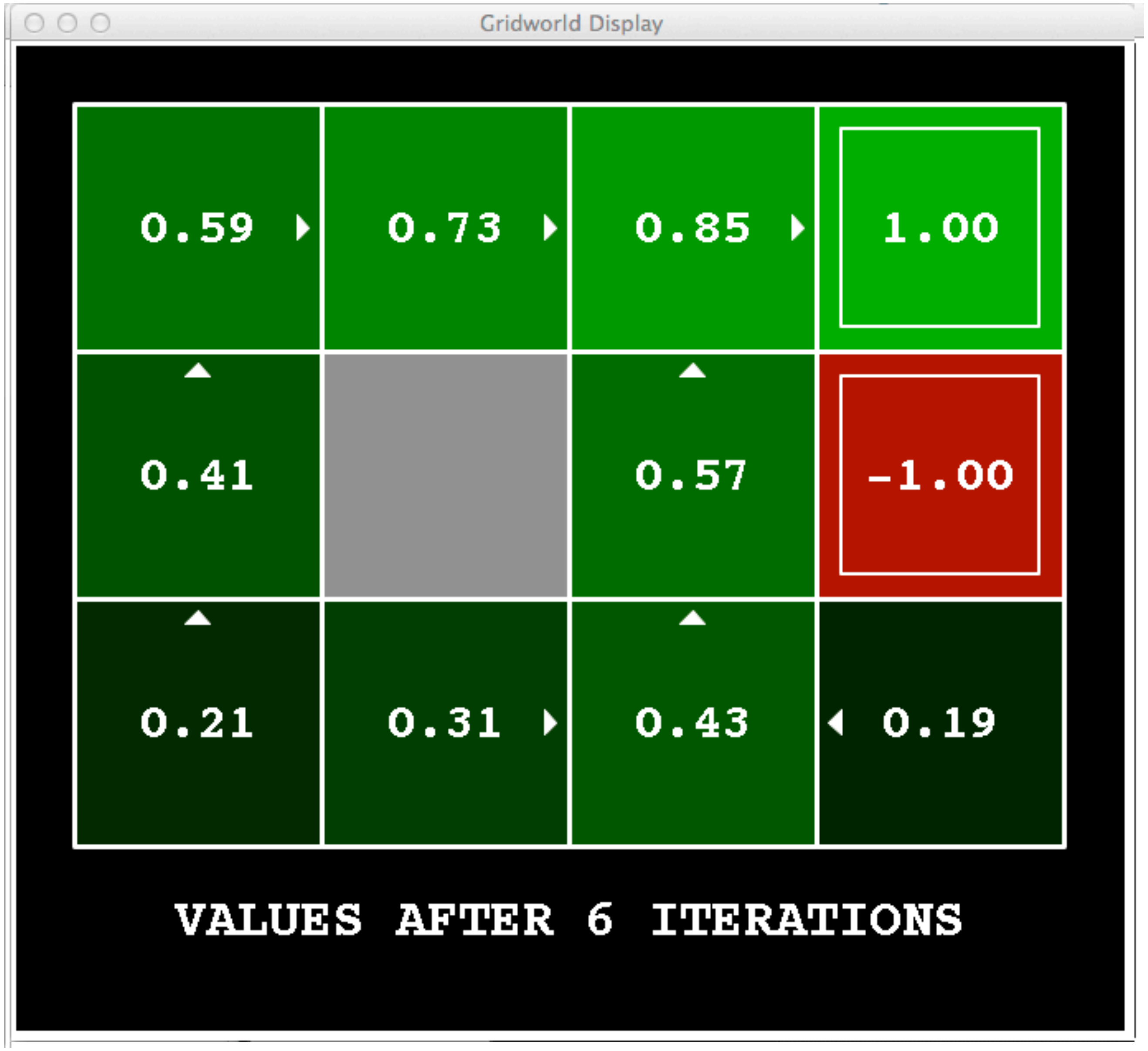
Value iteration example



0.51 ▶	0.72 ▶	0.84 ▶	1.00
▲ 0.27		▲ 0.55	-1.00
▲ 0.00	0.22 ▶	▲ 0.37	◀ 0.13

VALUES AFTER 5 ITERATIONS

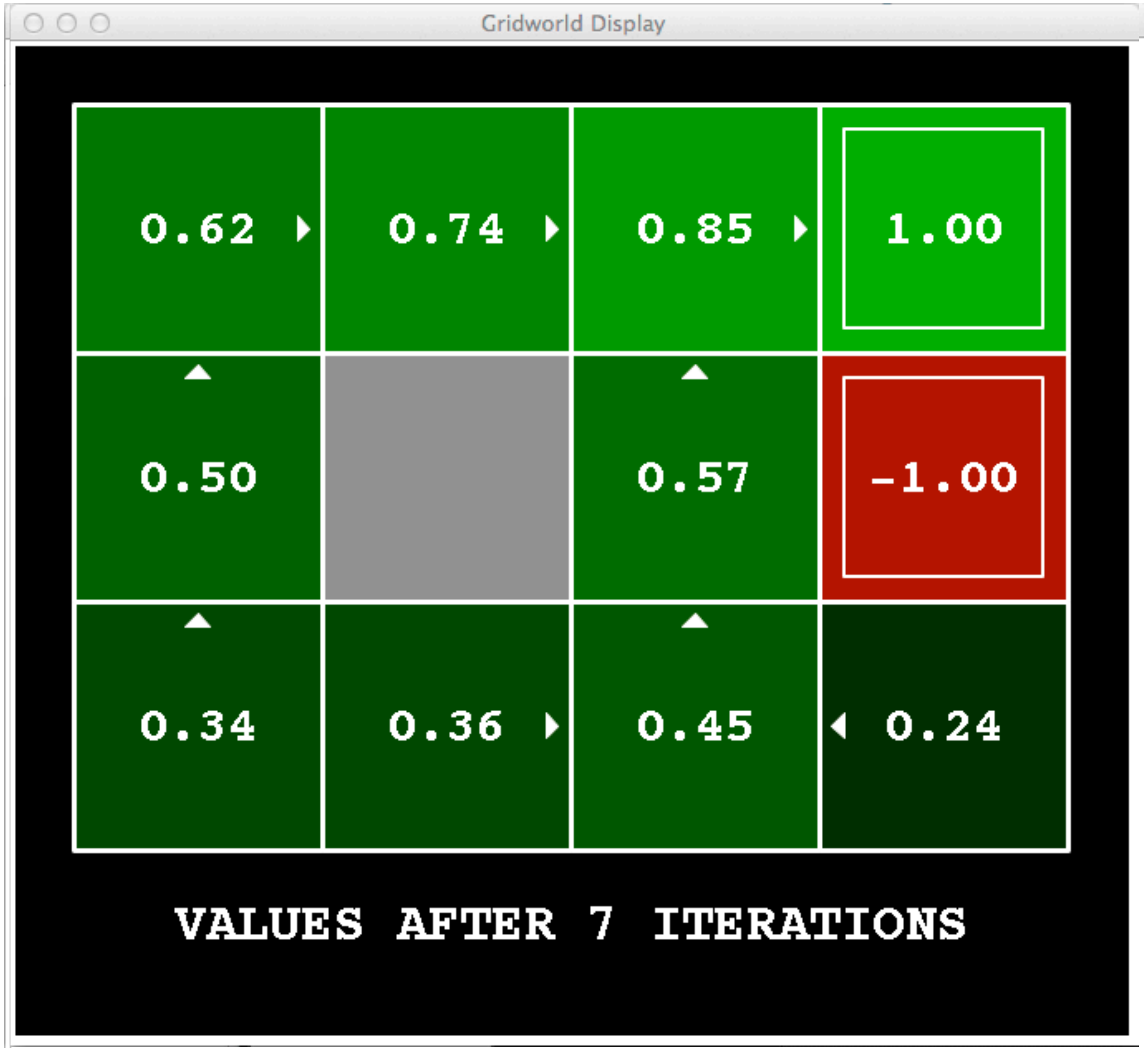
Value iteration example



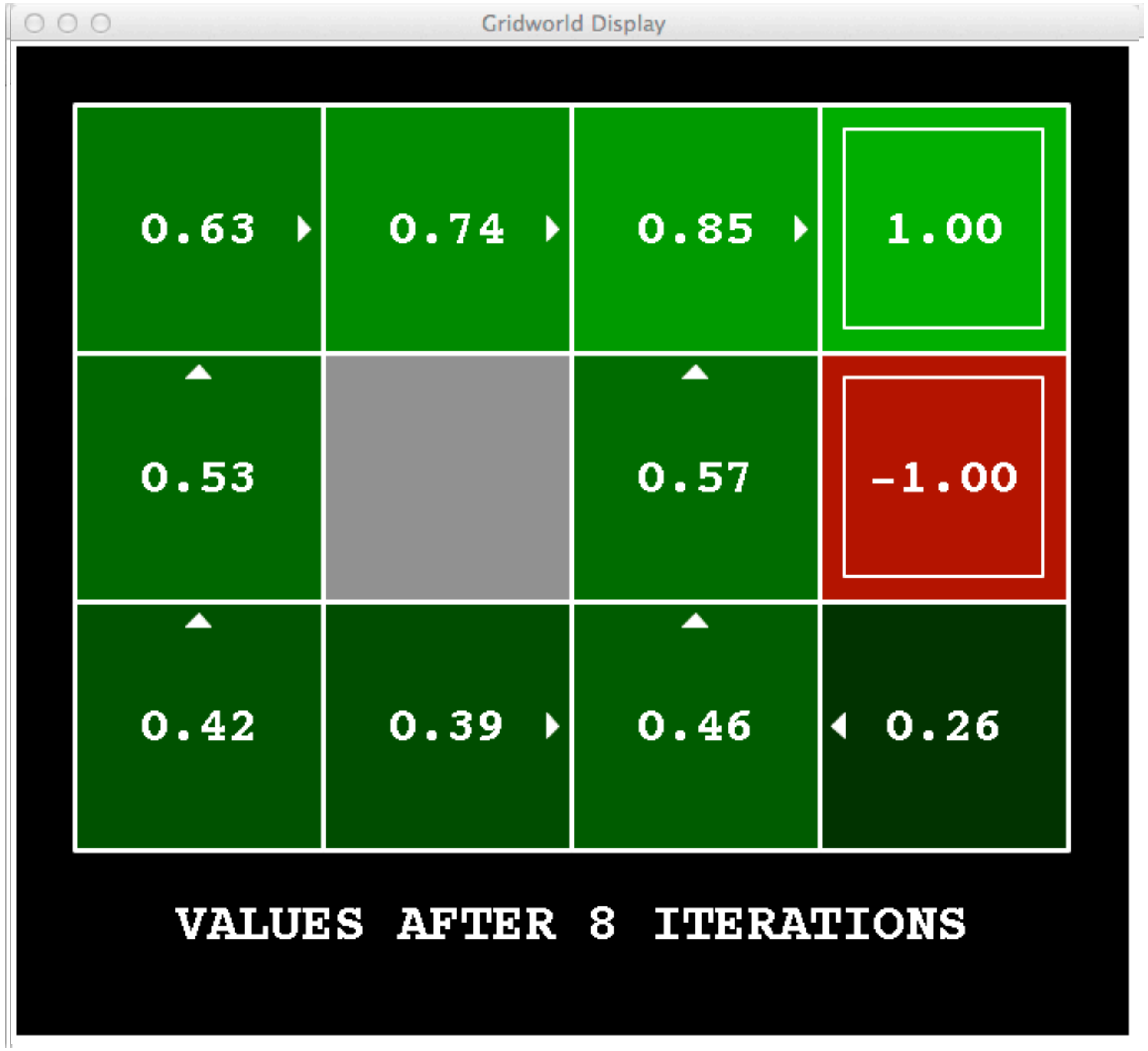
0.59 ▶	0.73 ▶	0.85 ▶	1.00
▲ 0.41		▲ 0.57	-1.00
▲ 0.21	0.31 ▶	▲ 0.43	◀ 0.19

VALUES AFTER 6 ITERATIONS

Value iteration example



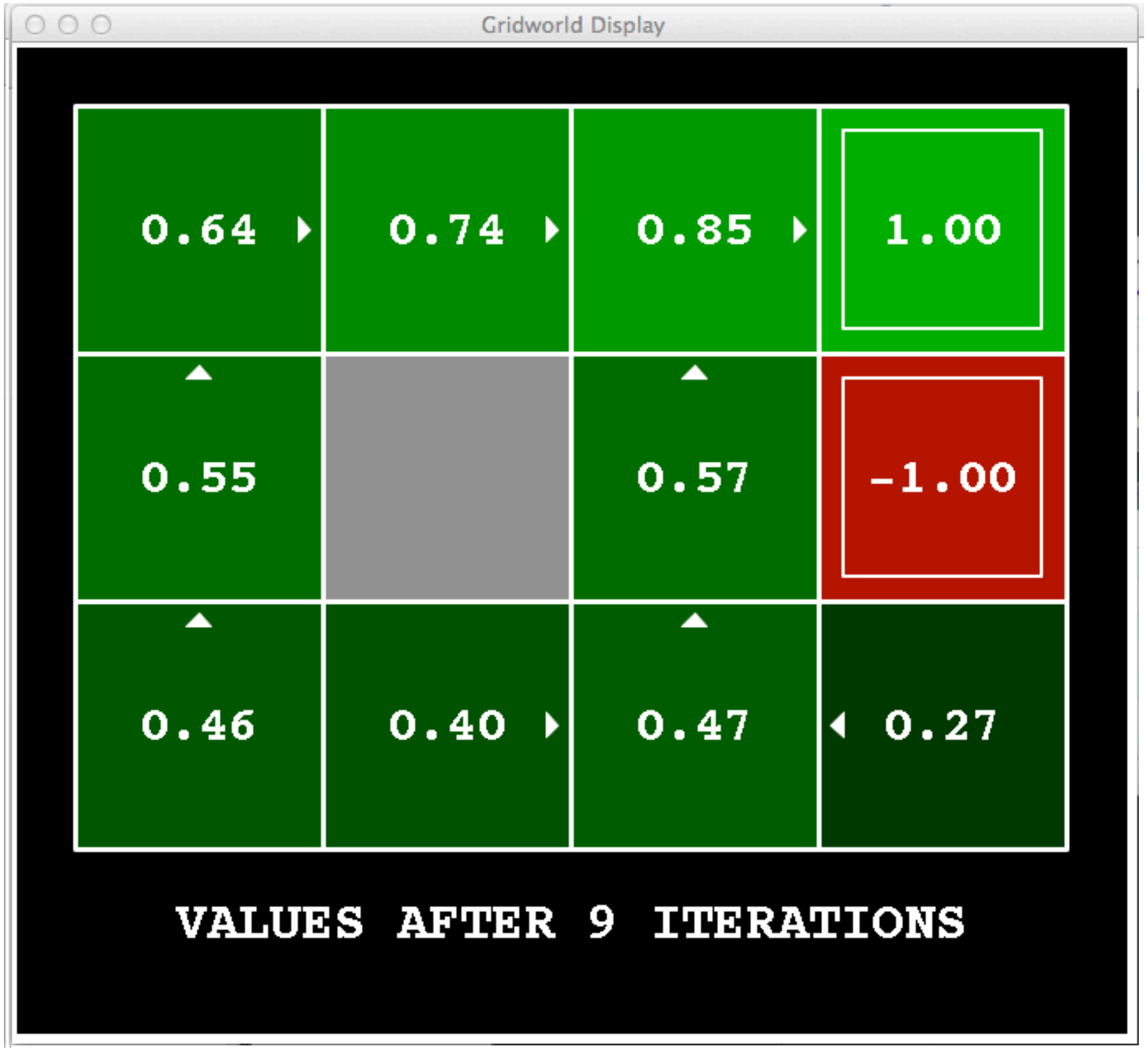
Value iteration example



0.63 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.53		▲ 0.57	-1.00
▲ 0.42	0.39 ▶	▲ 0.46	◀ 0.26

VALUES AFTER 8 ITERATIONS

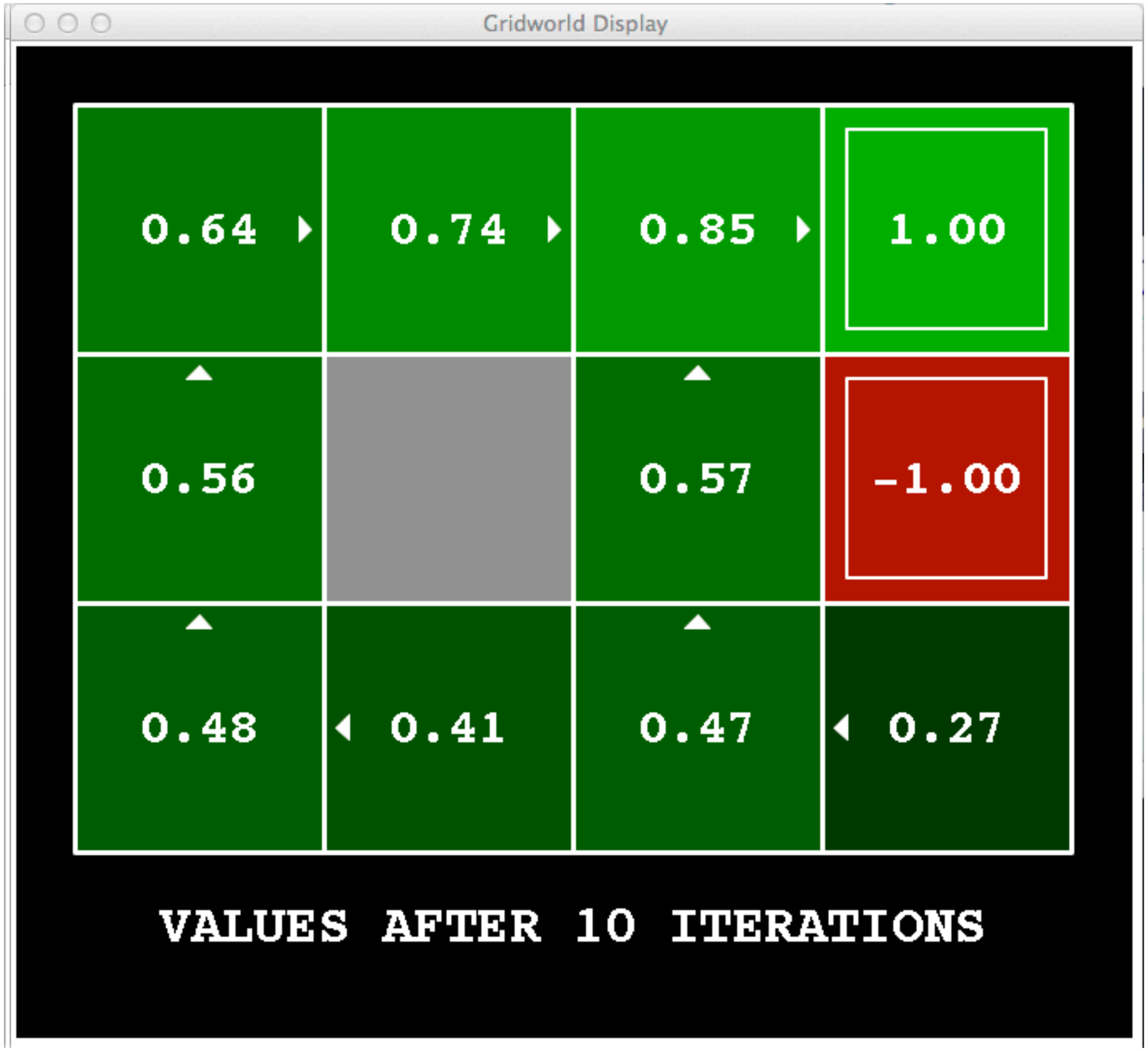
Value iteration example



0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.55		▴ 0.57	-1.00
▴ 0.46	0.40 ▸	▴ 0.47	◂ 0.27

VALUES AFTER 9 ITERATIONS

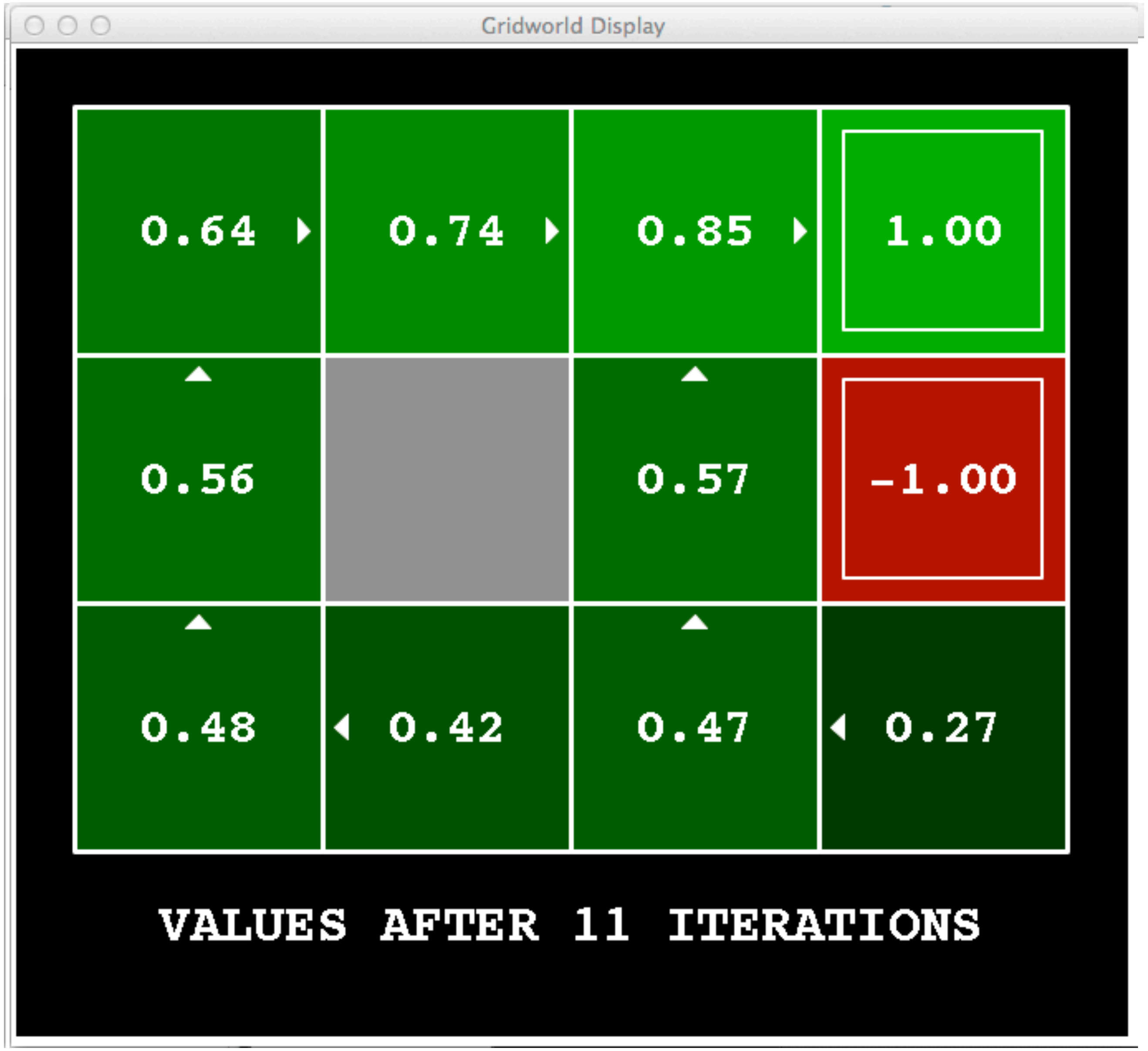
Value iteration example



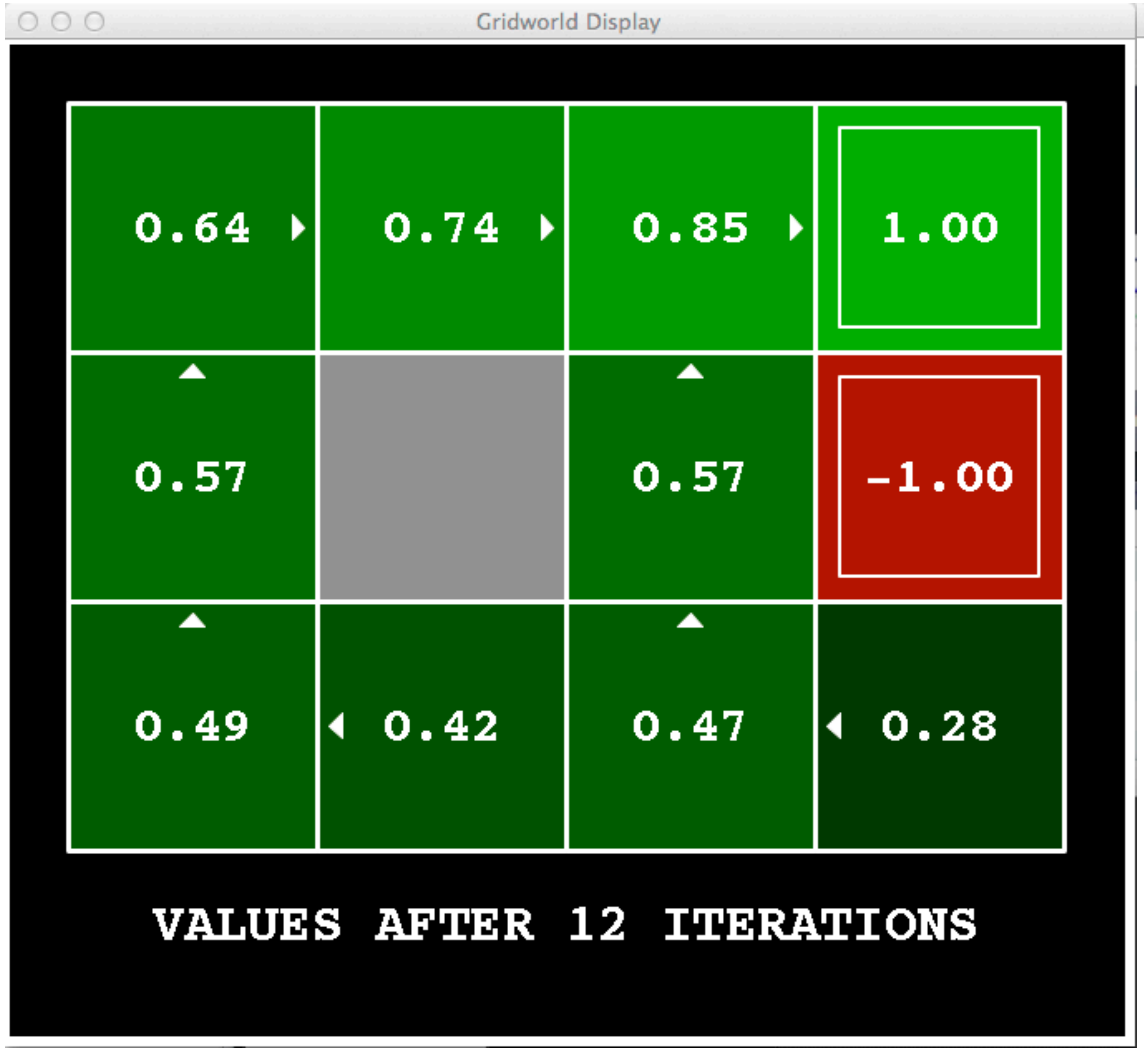
0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.56		▲ 0.57	-1.00
▲ 0.48	◀ 0.41	▲ 0.47	◀ 0.27

VALUES AFTER 10 ITERATIONS

Value iteration example

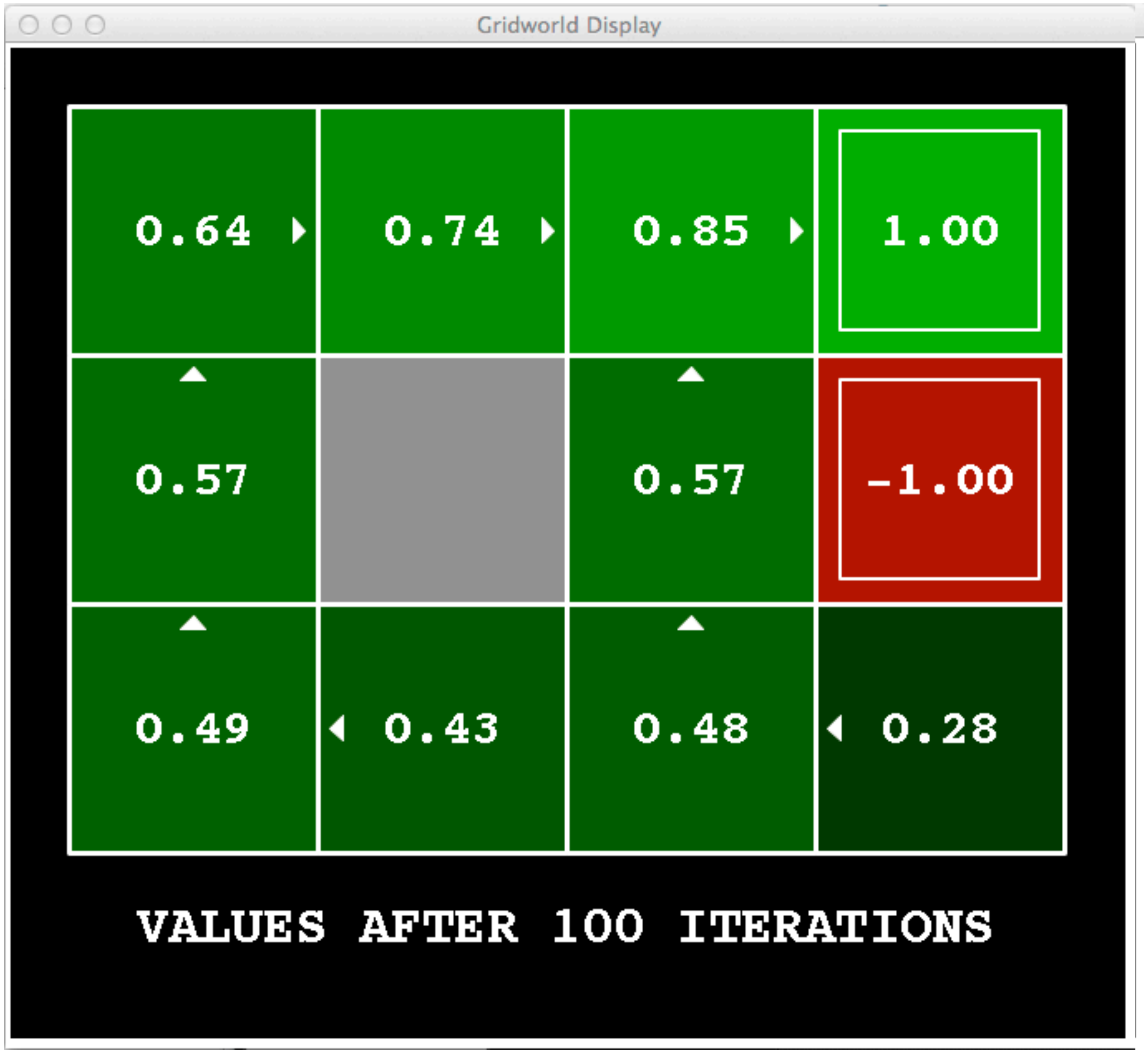


Value iteration example



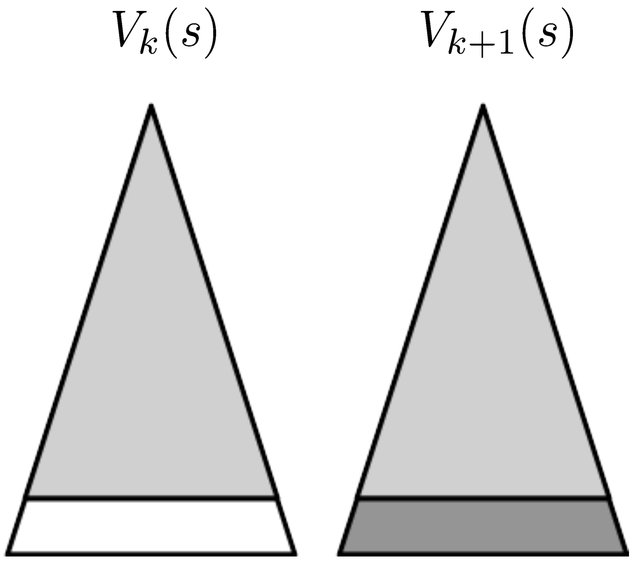
VALUES AFTER 12 ITERATIONS

Value iteration example



Proof sketch: convergence of value iteration

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



Bellman Equations and 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

Gauss-Siedel value iteration

Regular value iteration must maintain two arrays (the old U and the new U)

Gauss-Siedel value iteration only uses one array and can lead to faster convergence

Iterate through the state space and apply the Bellman update:

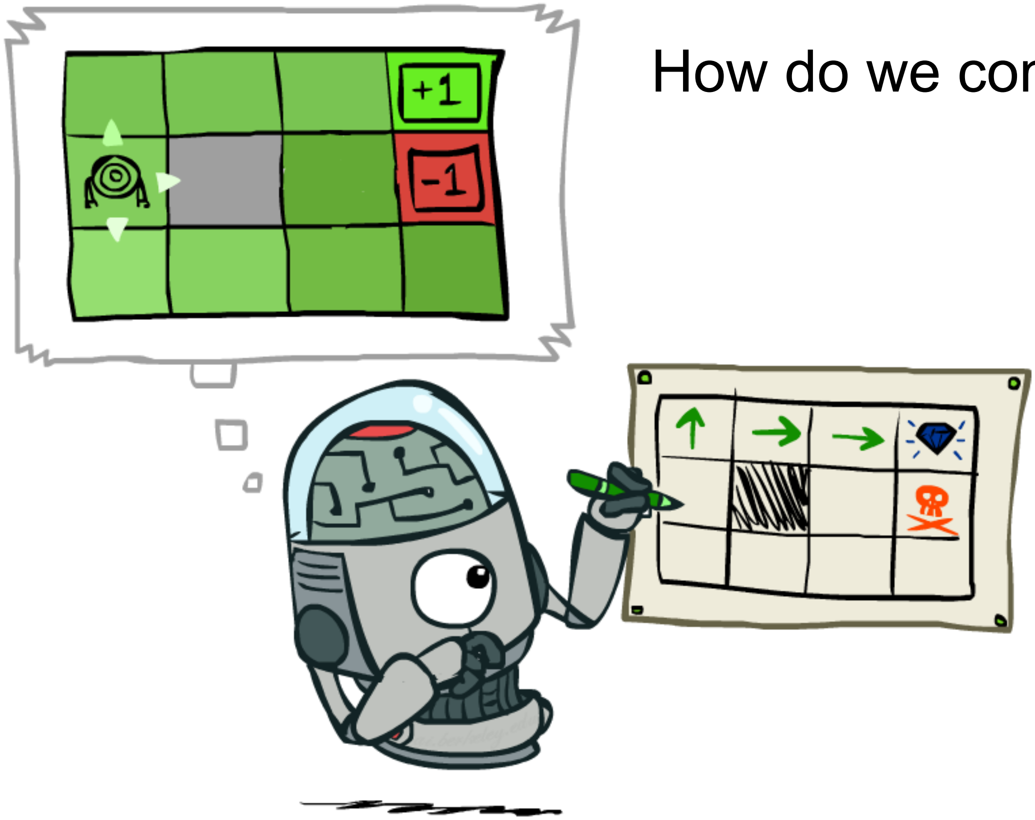
$$U(s) \leftarrow R(s) + \gamma \max_a \textstyle\sum_{s'} U(s')T(s,a,s')$$

Choice of ordering of updates can effect convergence rate

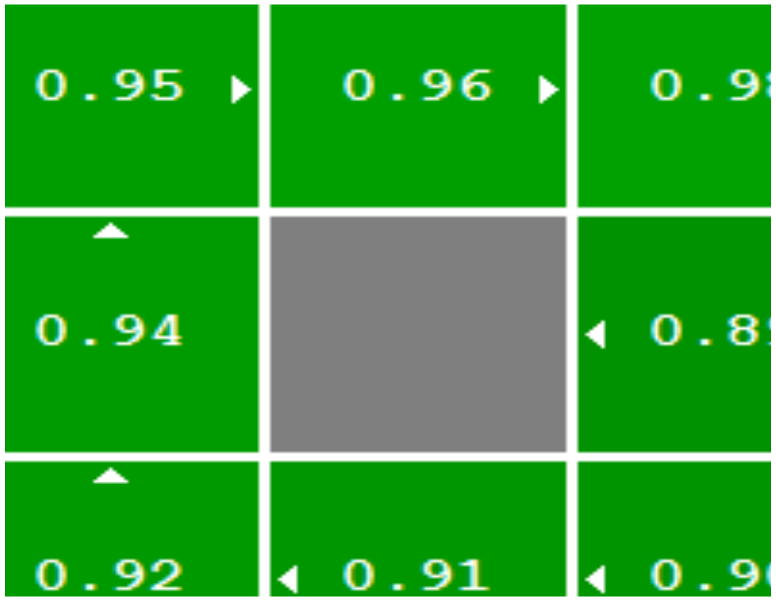
But, how do you compute a policy?

Suppose that we have run value iteration and now have a pretty good approximation of V* …

How do we compute the optimal policy?



But, how do you compute a policy?



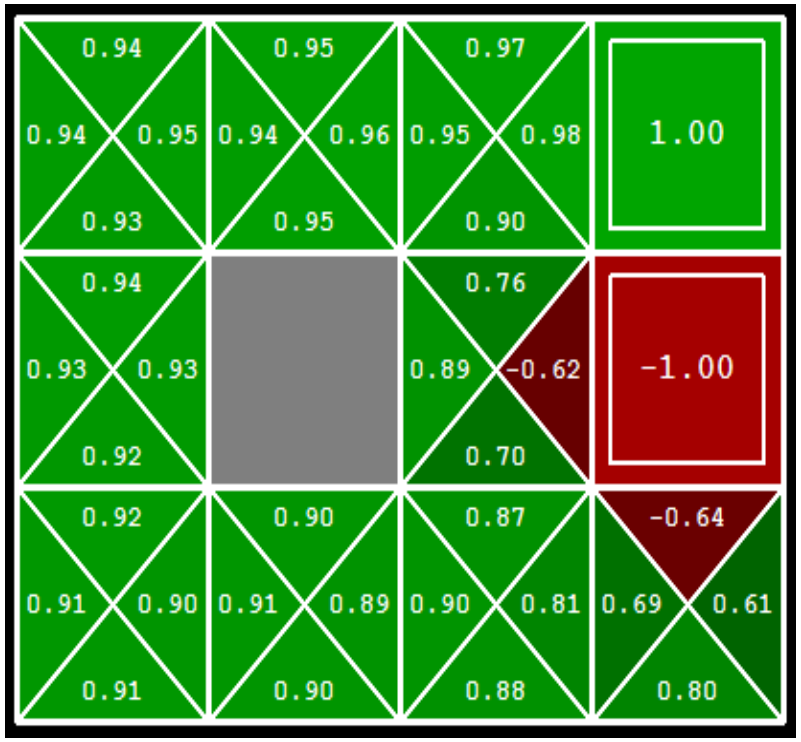
Given values calculated using value iteration, do one step of expectimax:

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

The optimal policy is implied by the optimal value function...

Computing actions from Q-values

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



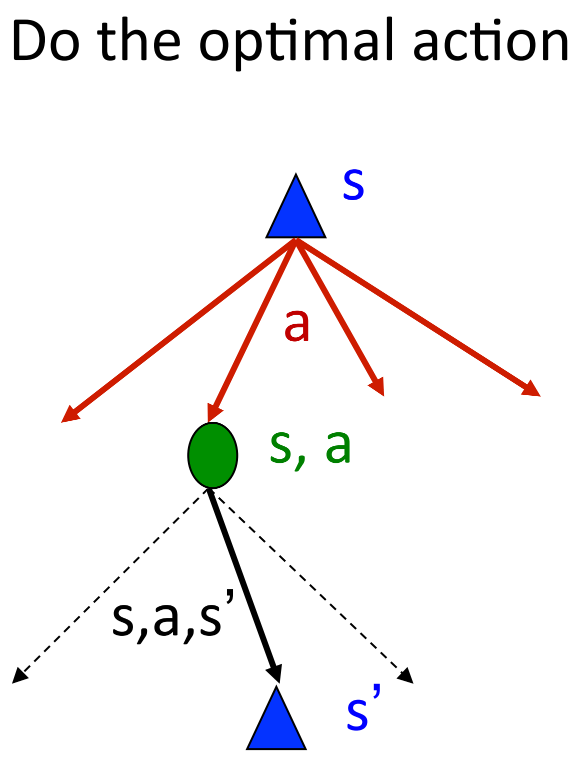
How should we act?

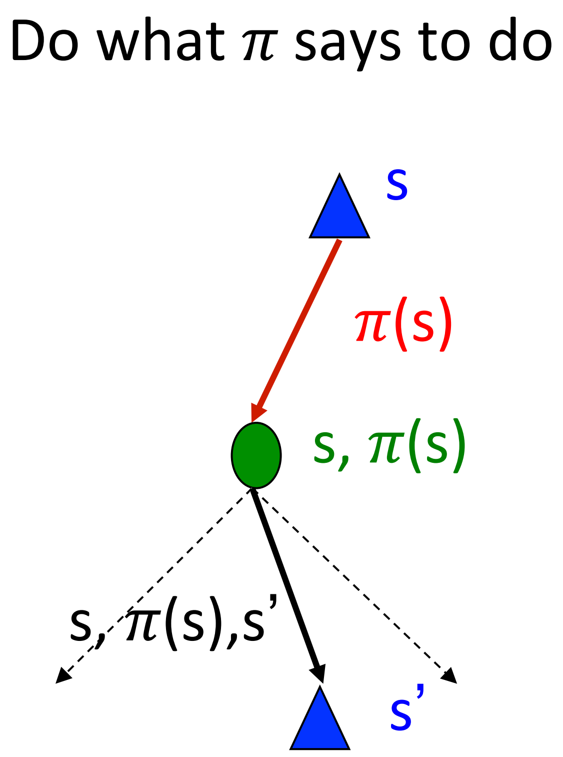
Completely trivial to decide!

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

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

Fixed policies





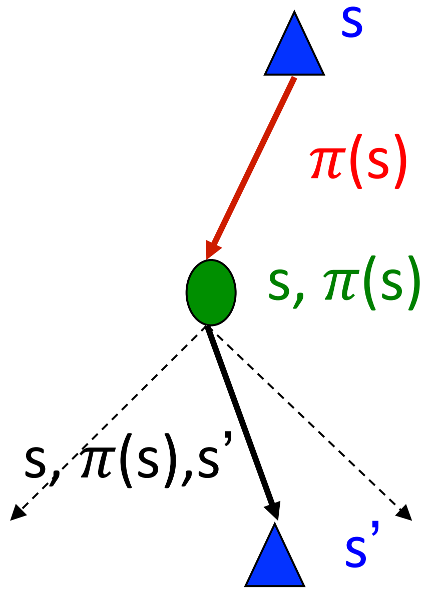
Expectimax trees max over all actions to compute the optimal values

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

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

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

Evaluating a fixed policy

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

Idea 1: Incrementally compute expected utility after *k* steps of executing π (like value iteration)

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

$$V_1^\pi(s) = R(s,\pi(s))$$

$$\ldots$$

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

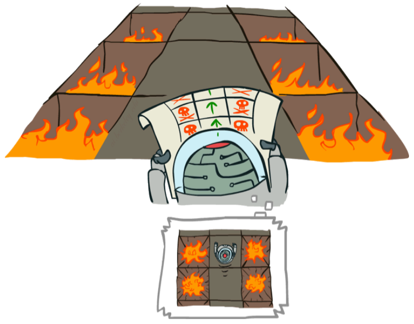
Dynamic programming as iterative evaluation

Efficiency: O(S^2) per iteration

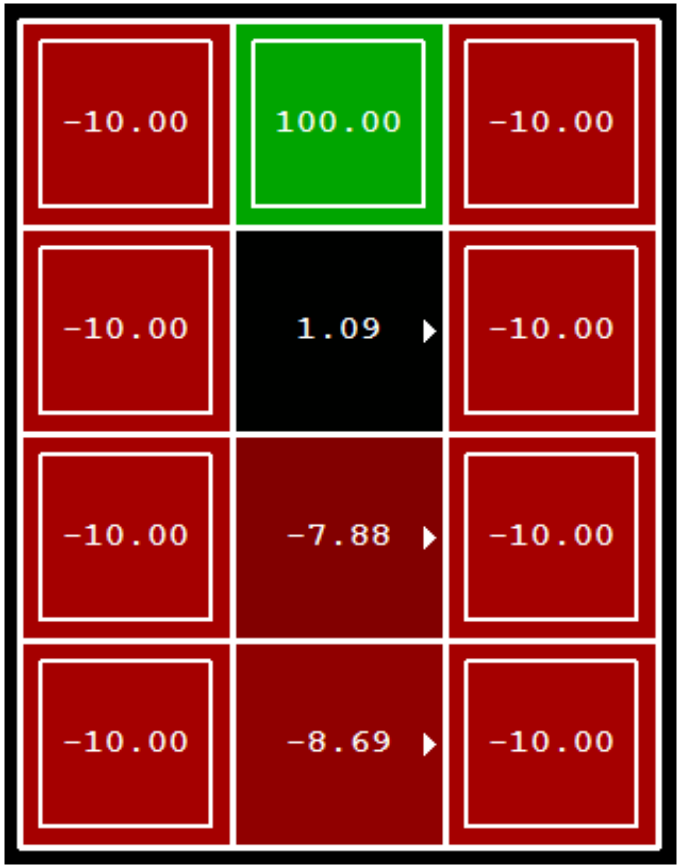
Idea 2: Without the maxes, the Bellman equations are just a linear system

Solve with Matlab (or your favorite linear system solver)

Example: policy evaluation

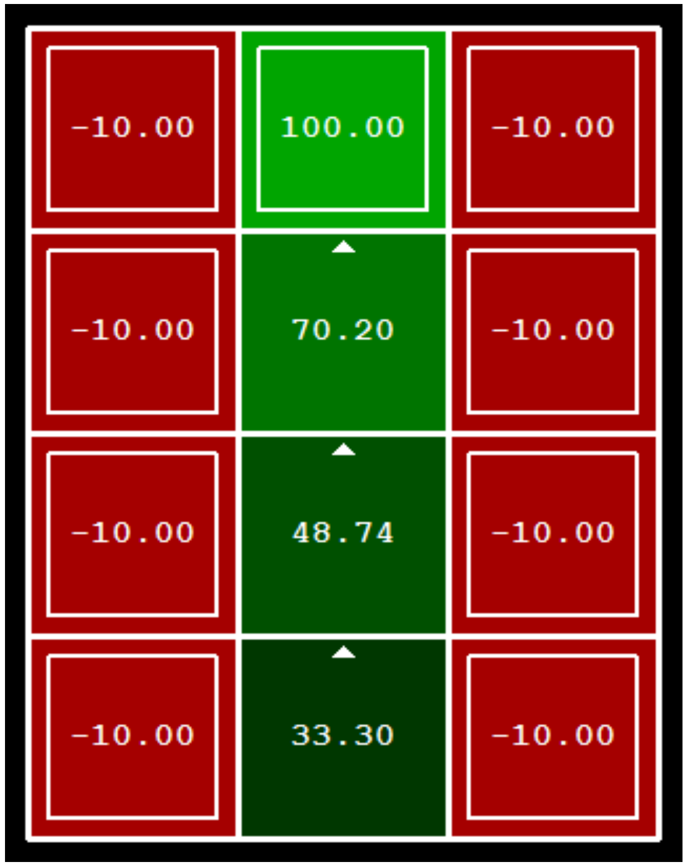


Always Go Right



-10.00	100.00	-10.00
-10.00	1.09 ▸	-10.00
-10.00	-7.88 ▸	-10.00
-10.00	-8.69 ▸	-10.00

Always Go Forward

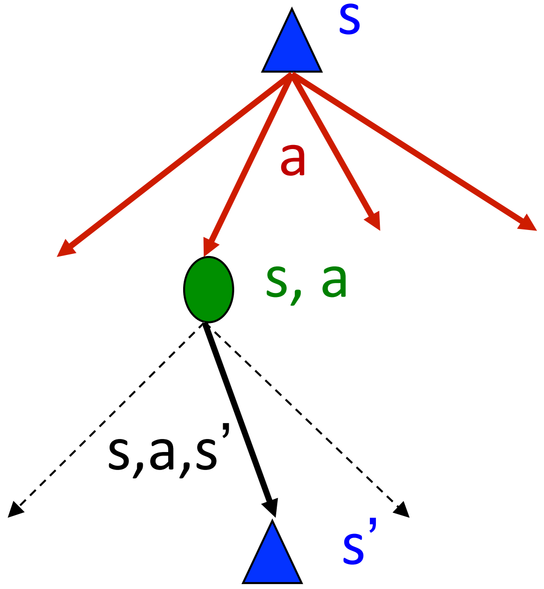


-10.00	100.00	-10.00
-10.00	▴ 70.20	-10.00
-10.00	▴ 48.74	-10.00
-10.00	▴ 33.30	-10.00

Problems with value iteration

Value iteration repeats the Bellman updates:

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

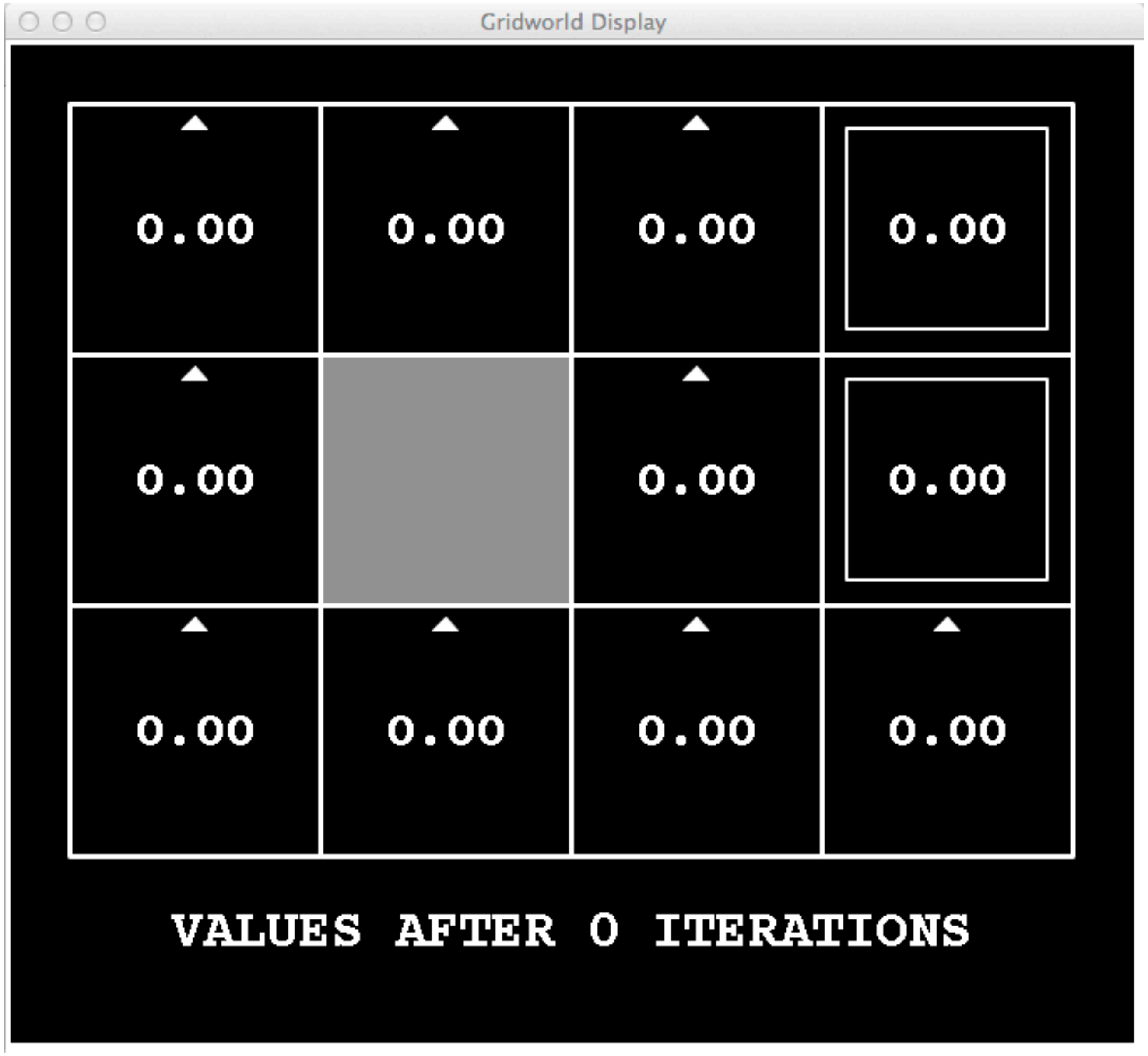


Problem 1: It's slow – O(S^2A) per iteration

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

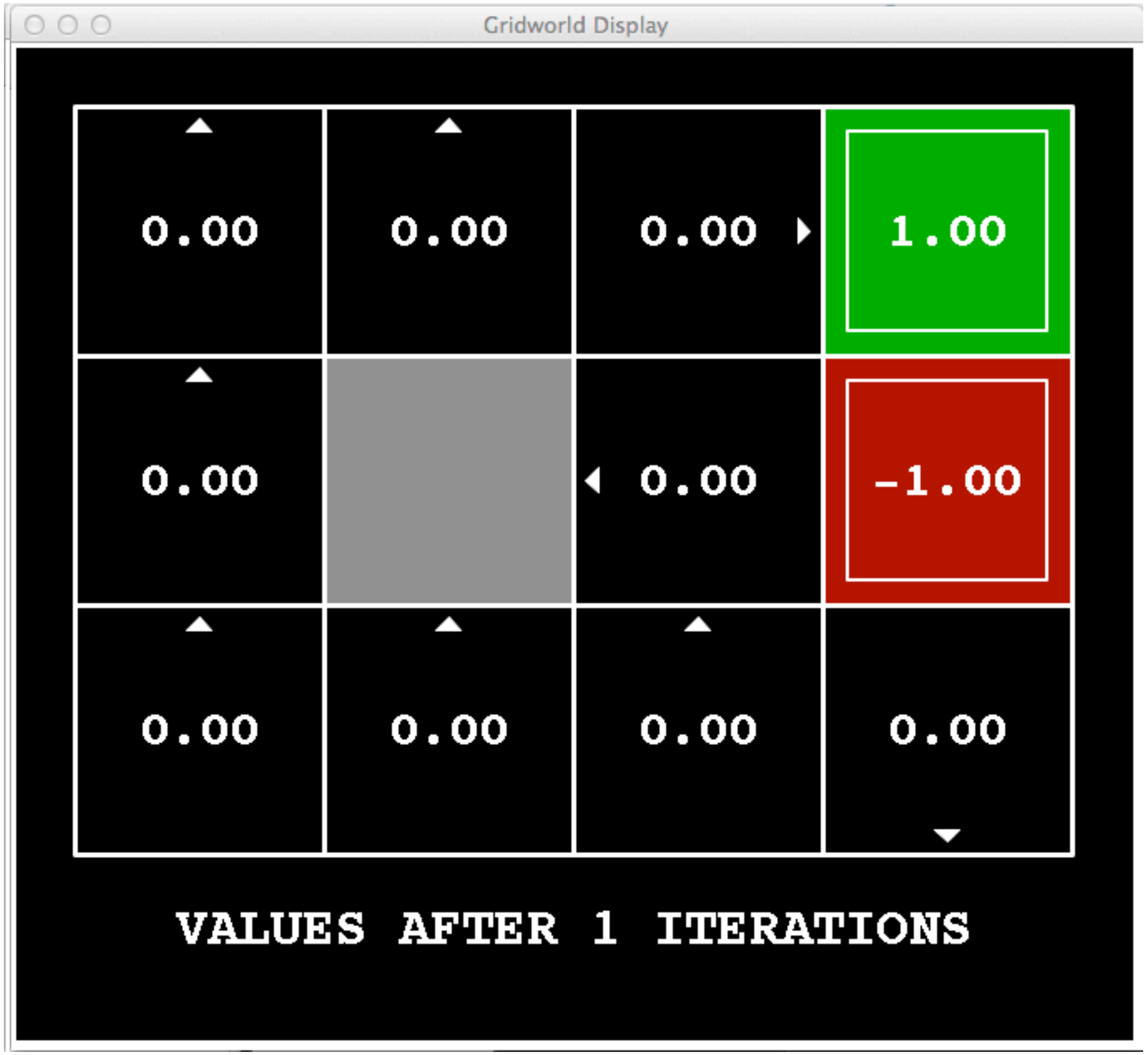
Problem 3: The policy often converges long before the values

Value iteration example



Noise = 0.2
Discount = 0.9
Living reward = 0

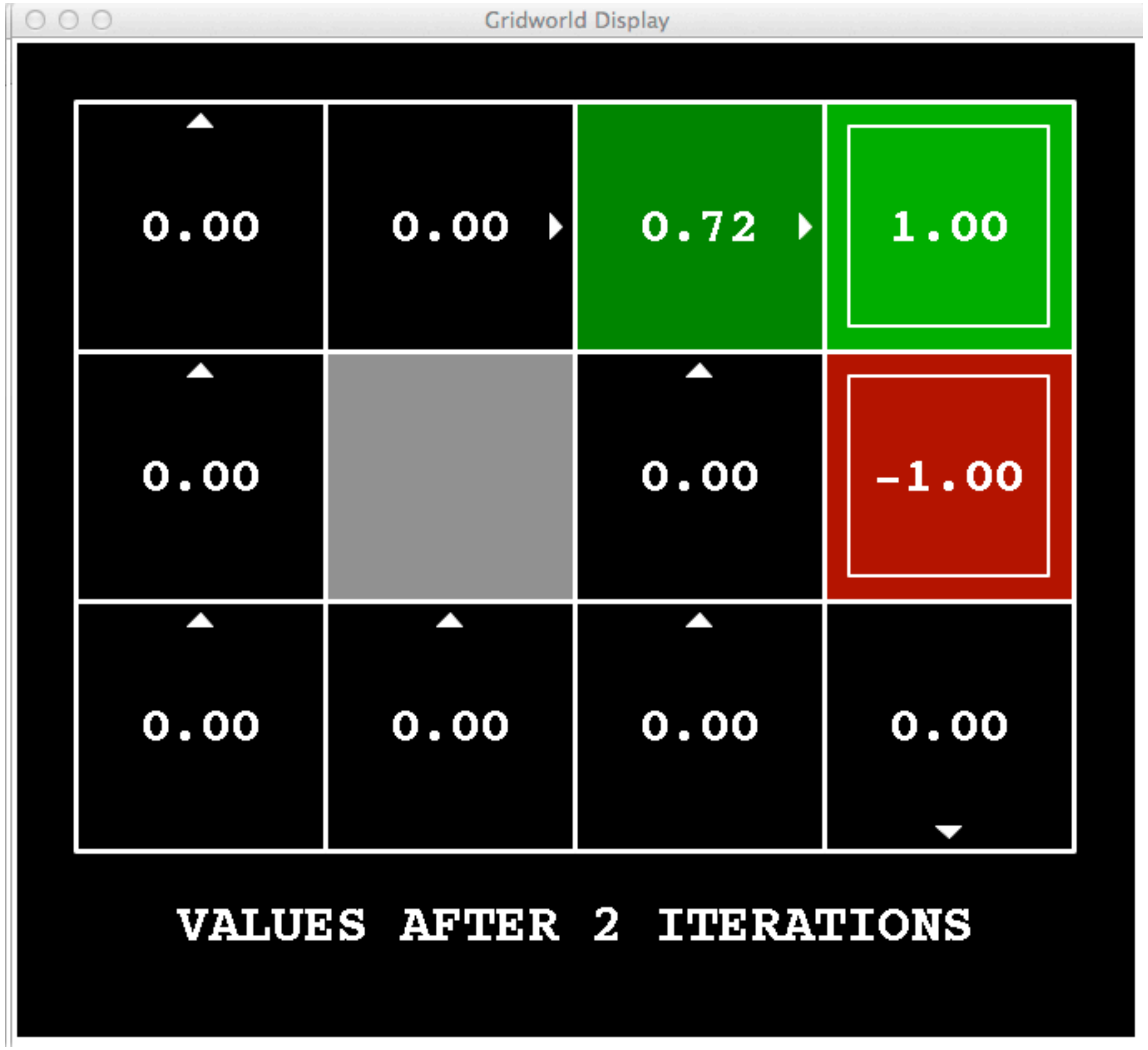
Value iteration example



0.00	0.00	0.00	1.00
0.00		0.00	-1.00
0.00	0.00	0.00	0.00

VALUES AFTER 1 ITERATIONS

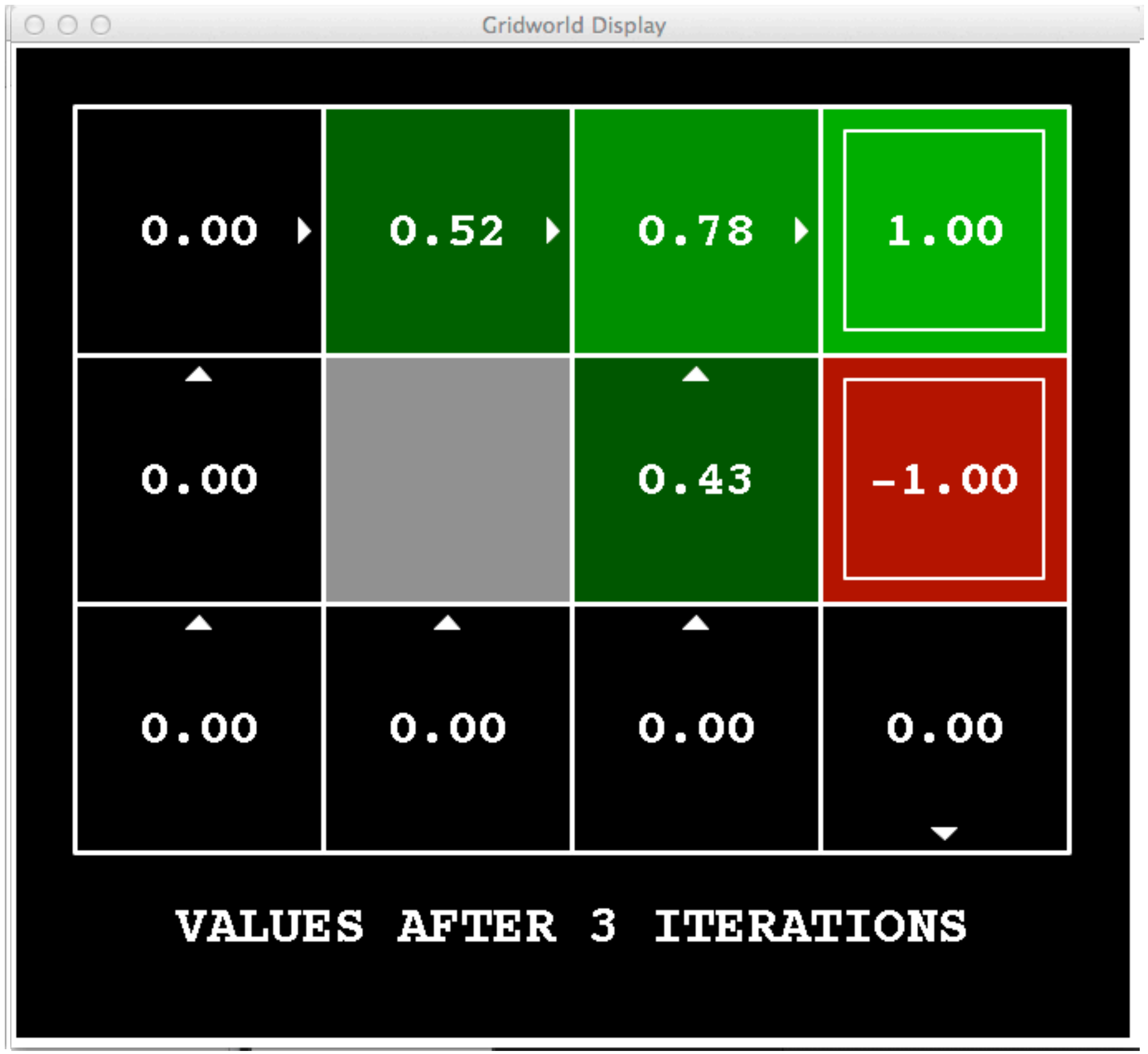
Value iteration example



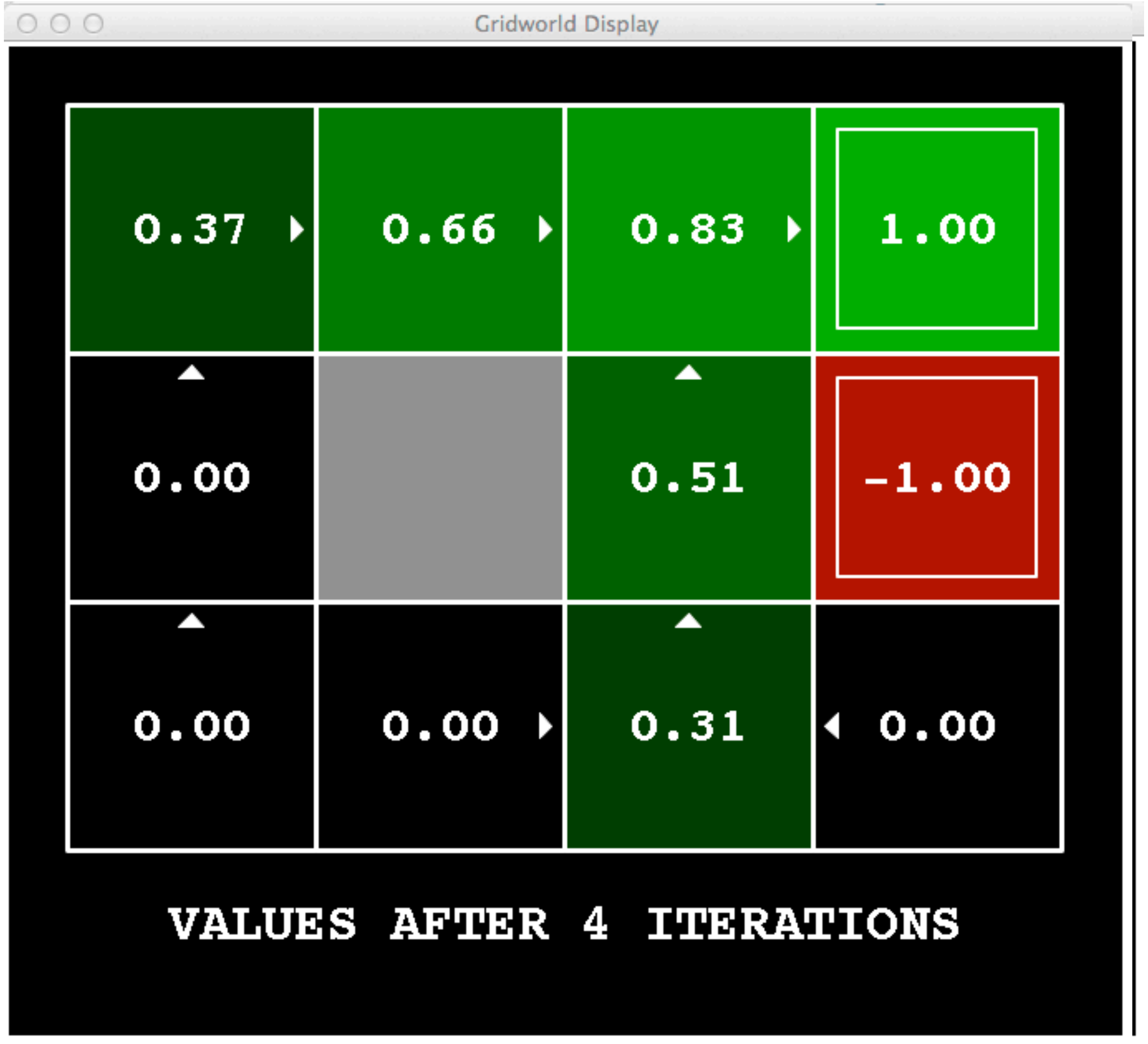
0.00 ▲	0.00 ▶	0.72 ▶	1.00
0.00 ▲		0.00 ▲	-1.00
0.00 ▲	0.00 ▲	0.00 ▲	0.00 ▼

VALUES AFTER 2 ITERATIONS

Value iteration example



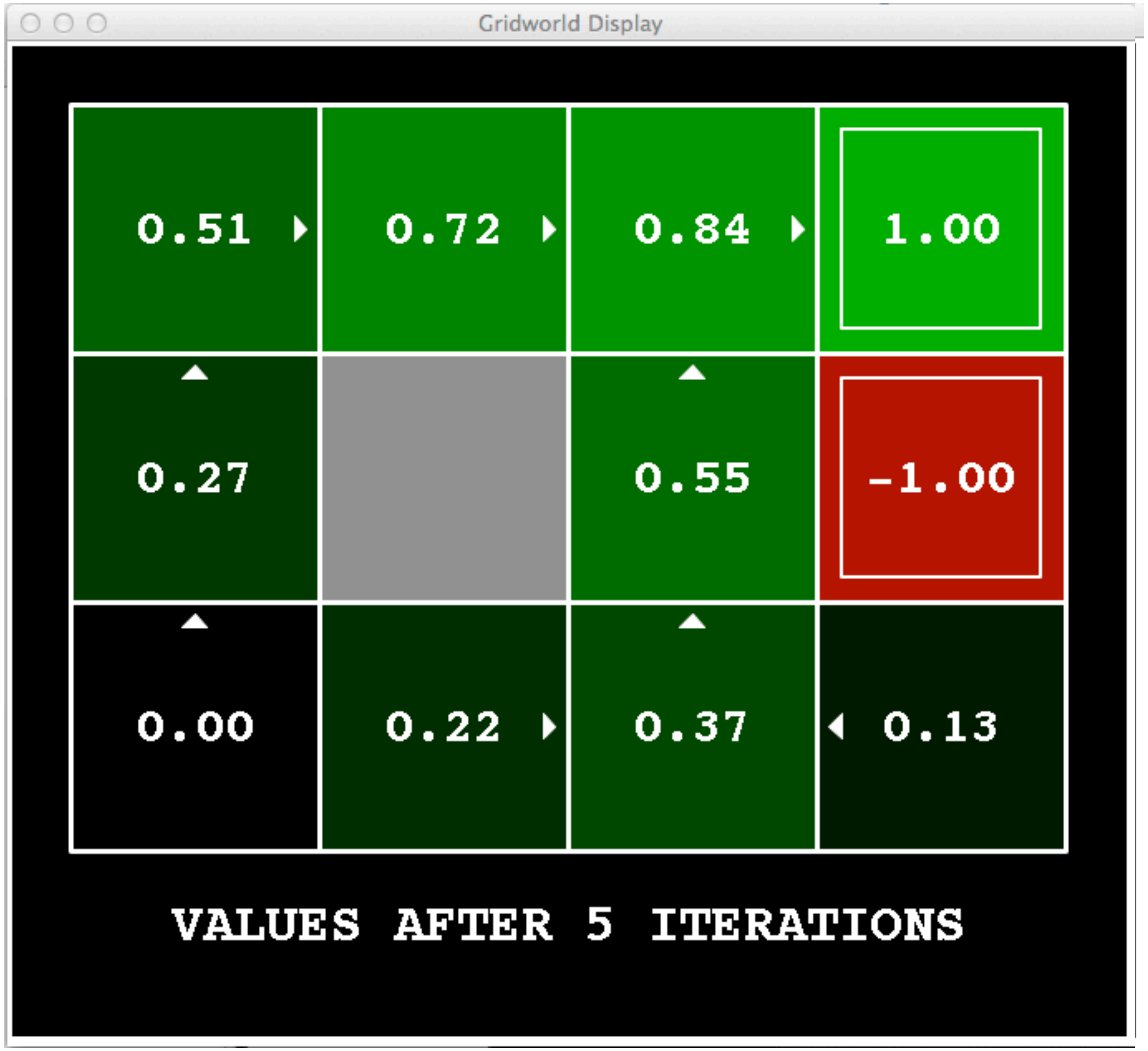
Value iteration example



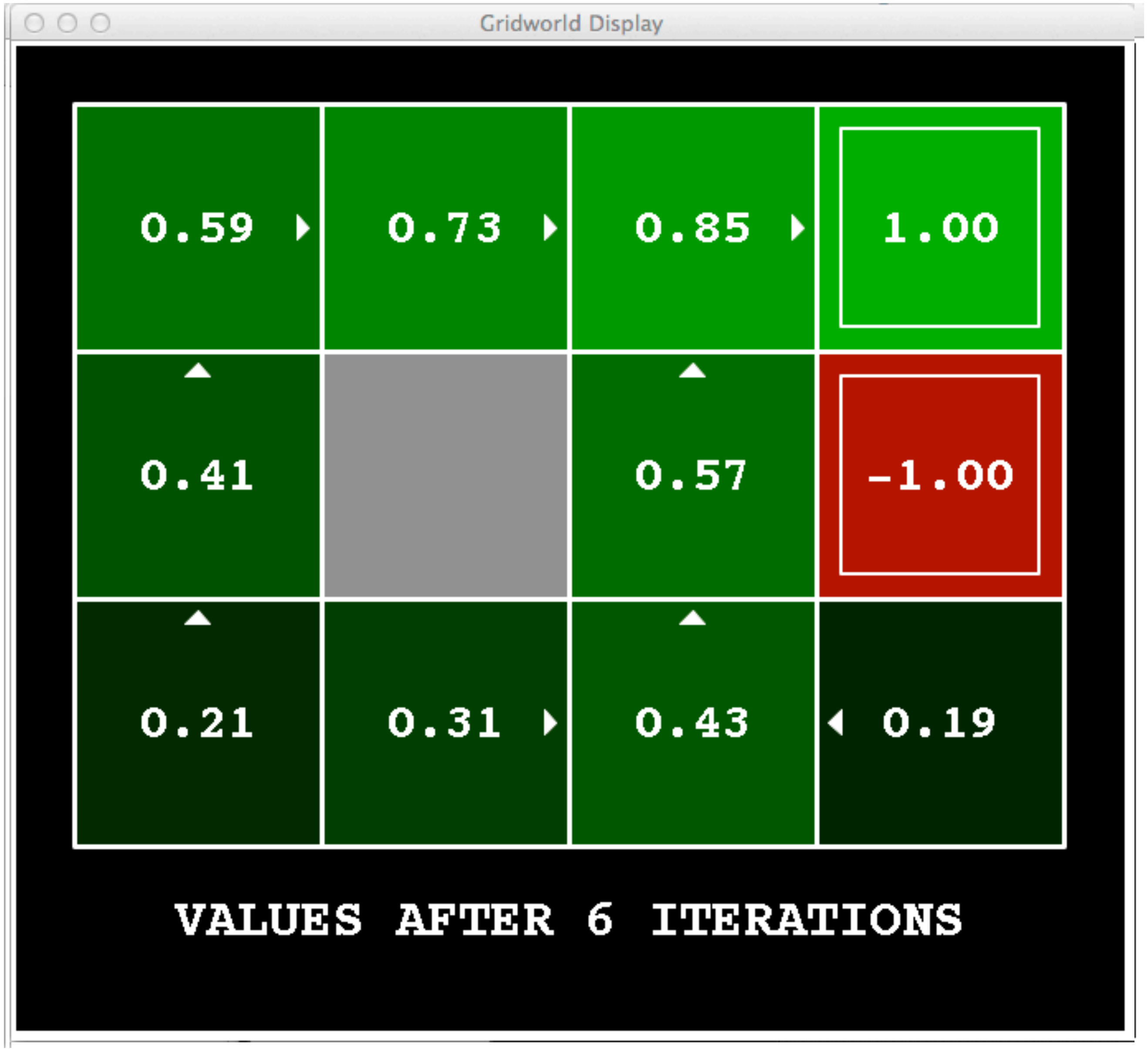
0.37 ▸	0.66 ▸	0.83 ▸	1.00
▴ 0.00		▴ 0.51	-1.00
▴ 0.00	0.00 ▸	▴ 0.31	◂ 0.00

VALUES AFTER 4 ITERATIONS

Value iteration example

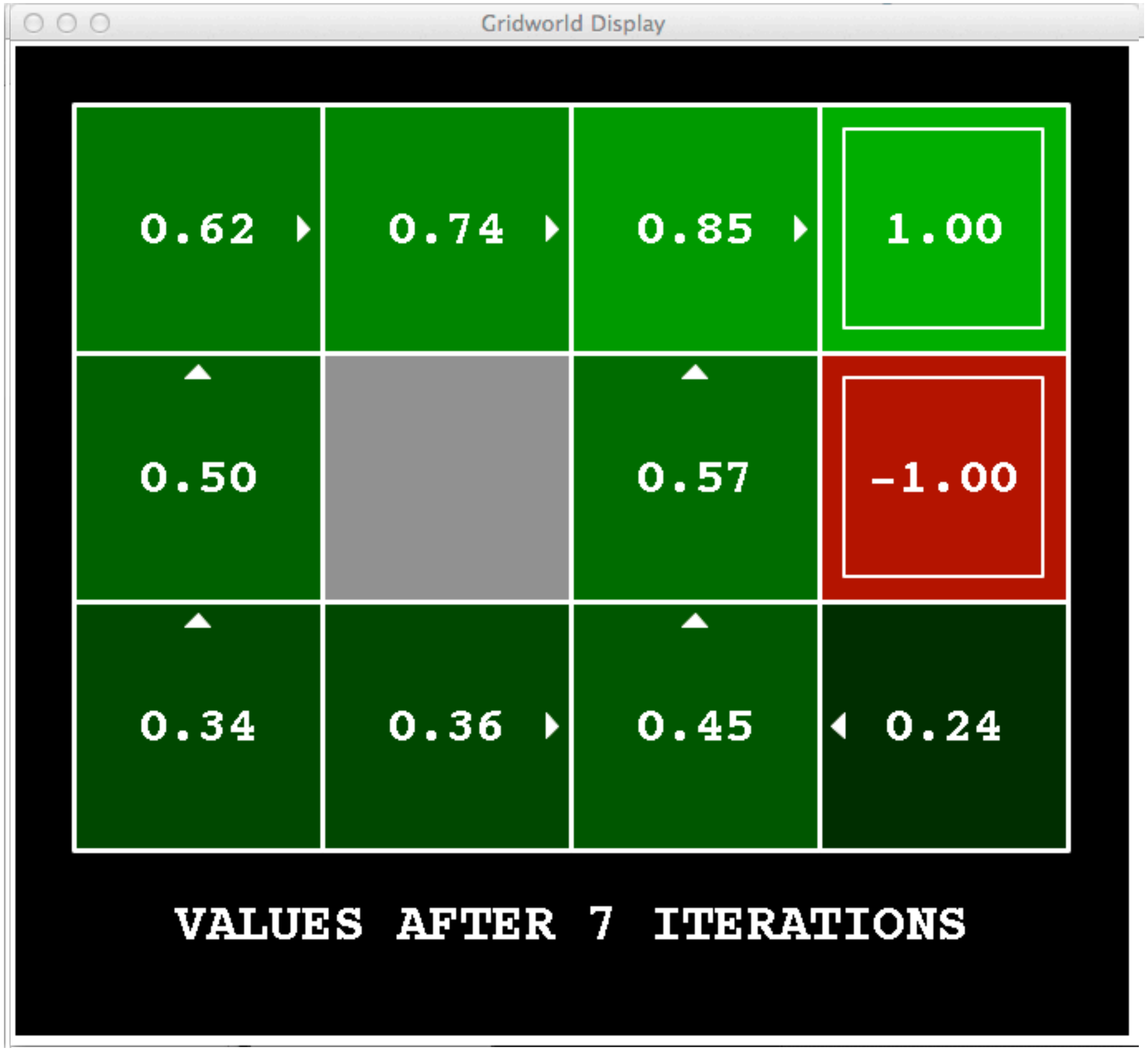


Value iteration example



VALUES AFTER 6 ITERATIONS

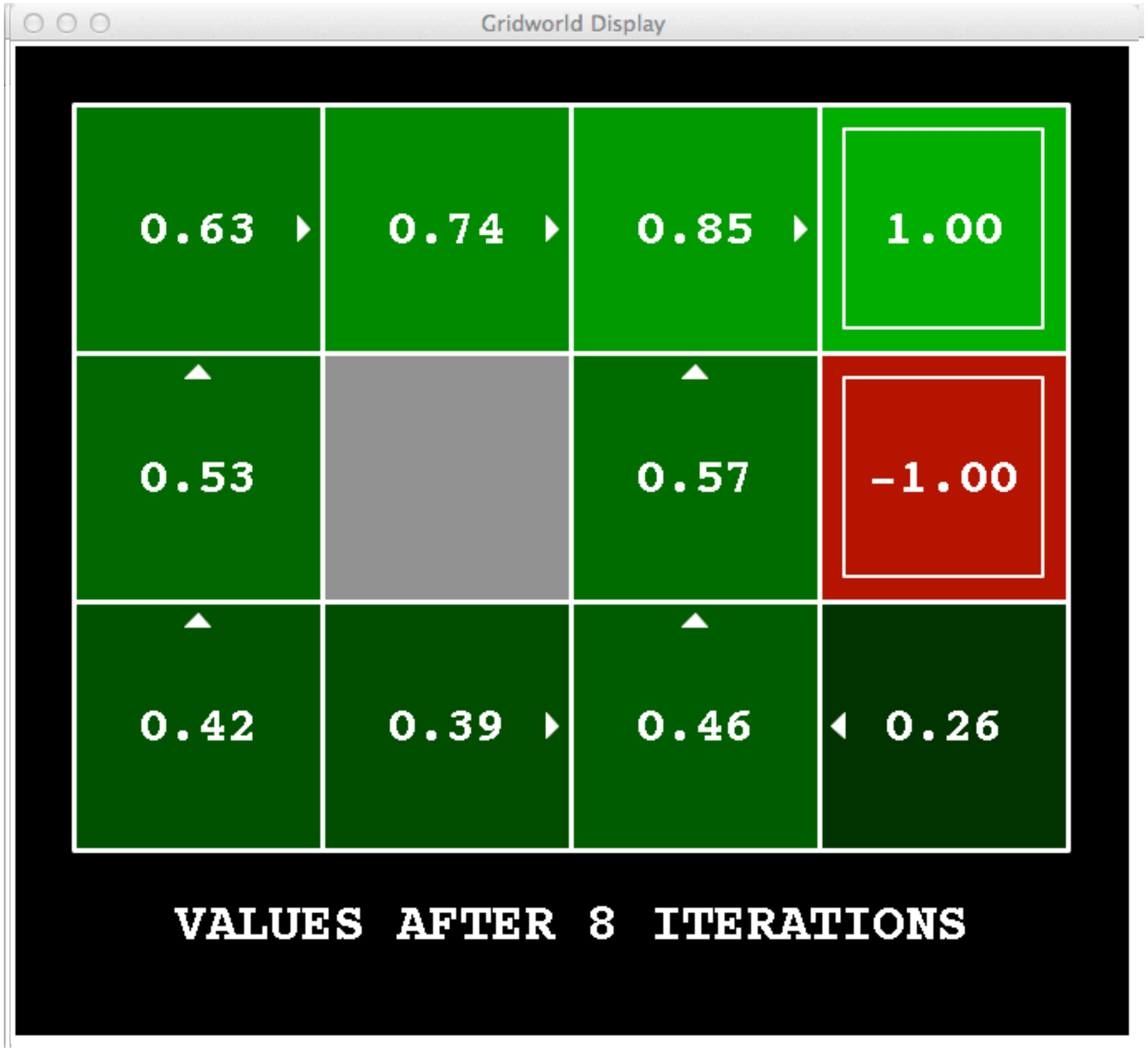
Value iteration example



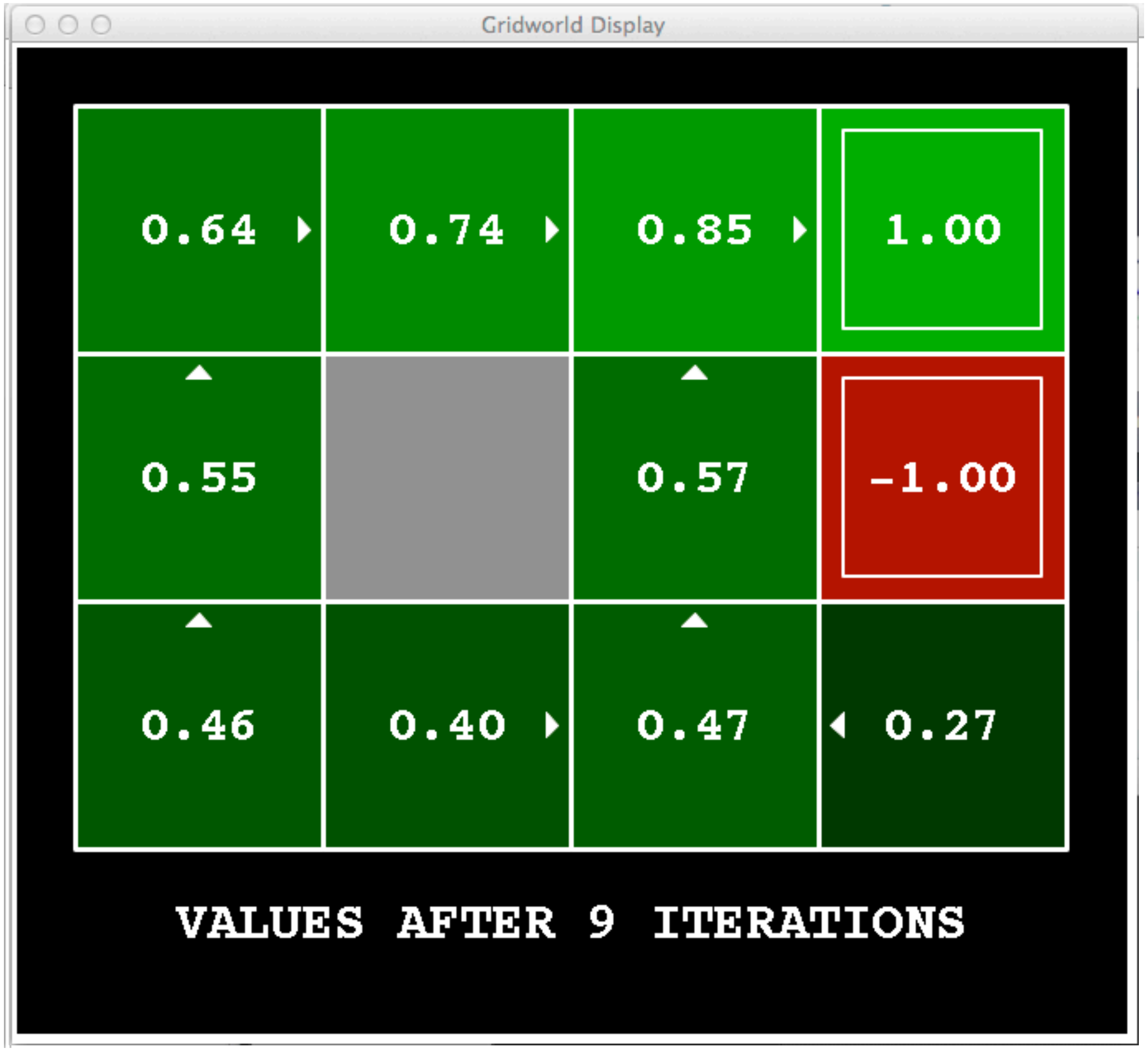
0.62 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.50		▲ 0.57	-1.00
▲ 0.34	0.36 ▶	▲ 0.45	◀ 0.24

VALUES AFTER 7 ITERATIONS

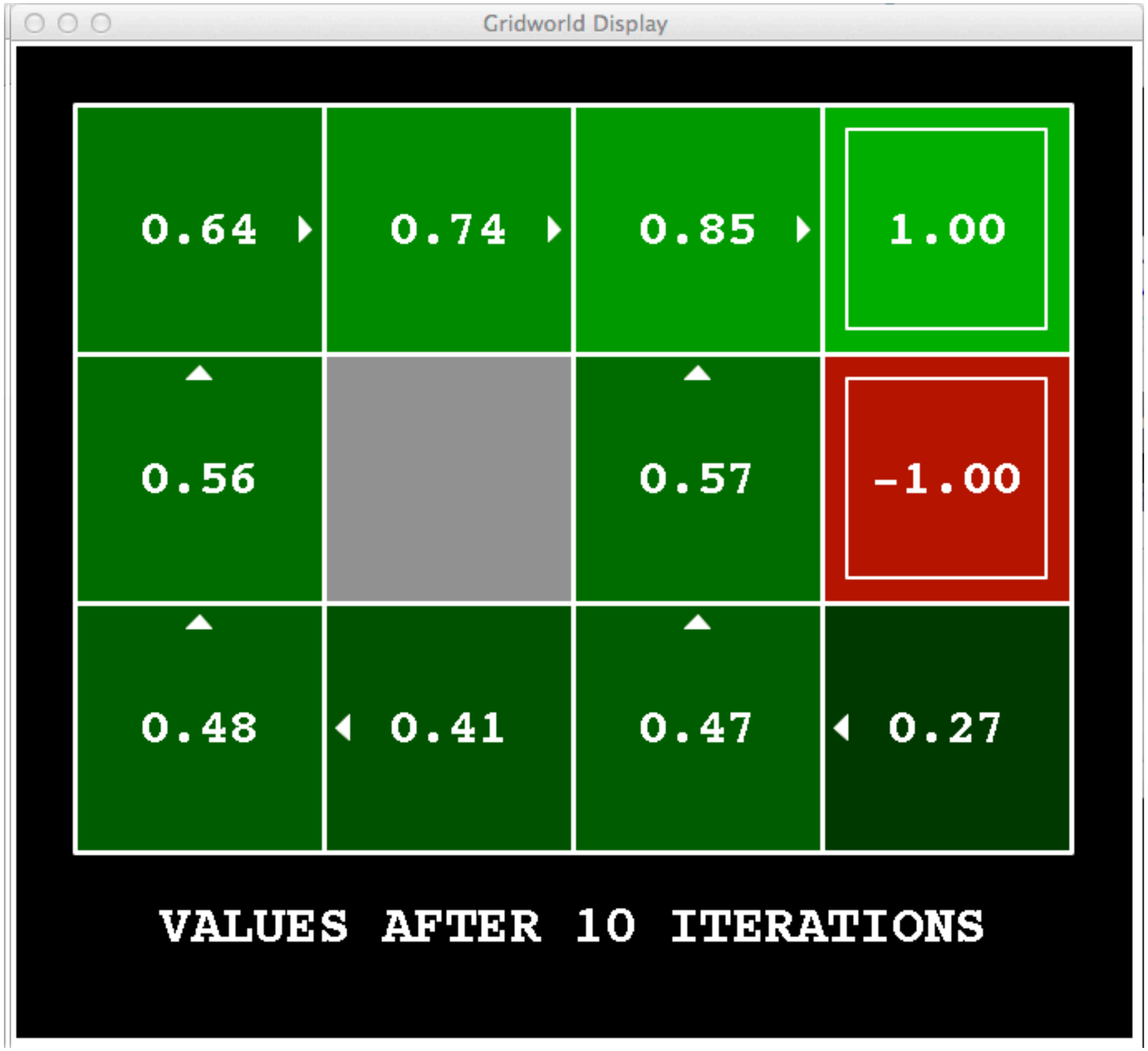
Value iteration example



Value iteration example



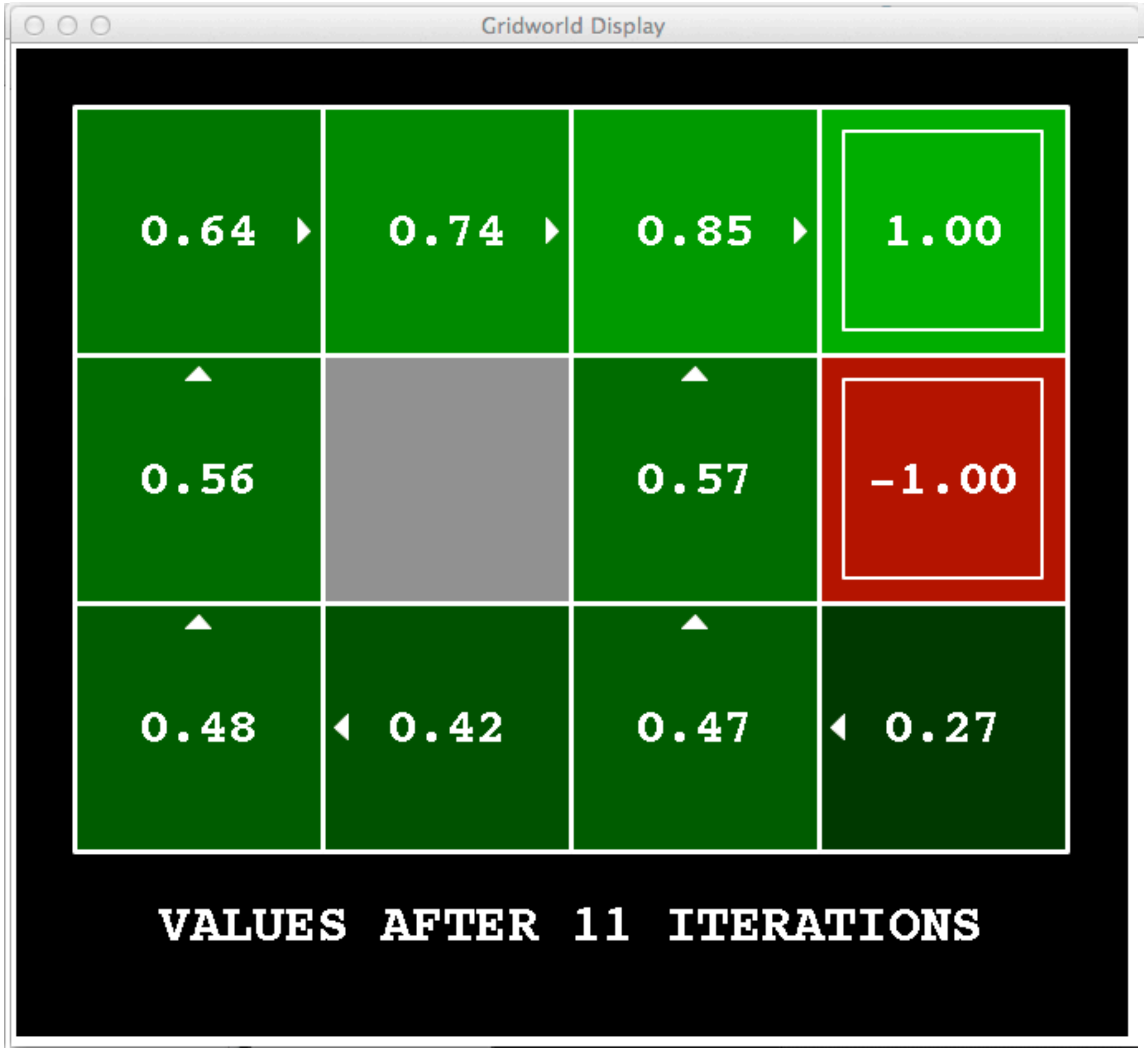
Value iteration example



0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.56		▴ 0.57	-1.00
▴ 0.48	◂ 0.41	▴ 0.47	◂ 0.27

VALUES AFTER 10 ITERATIONS

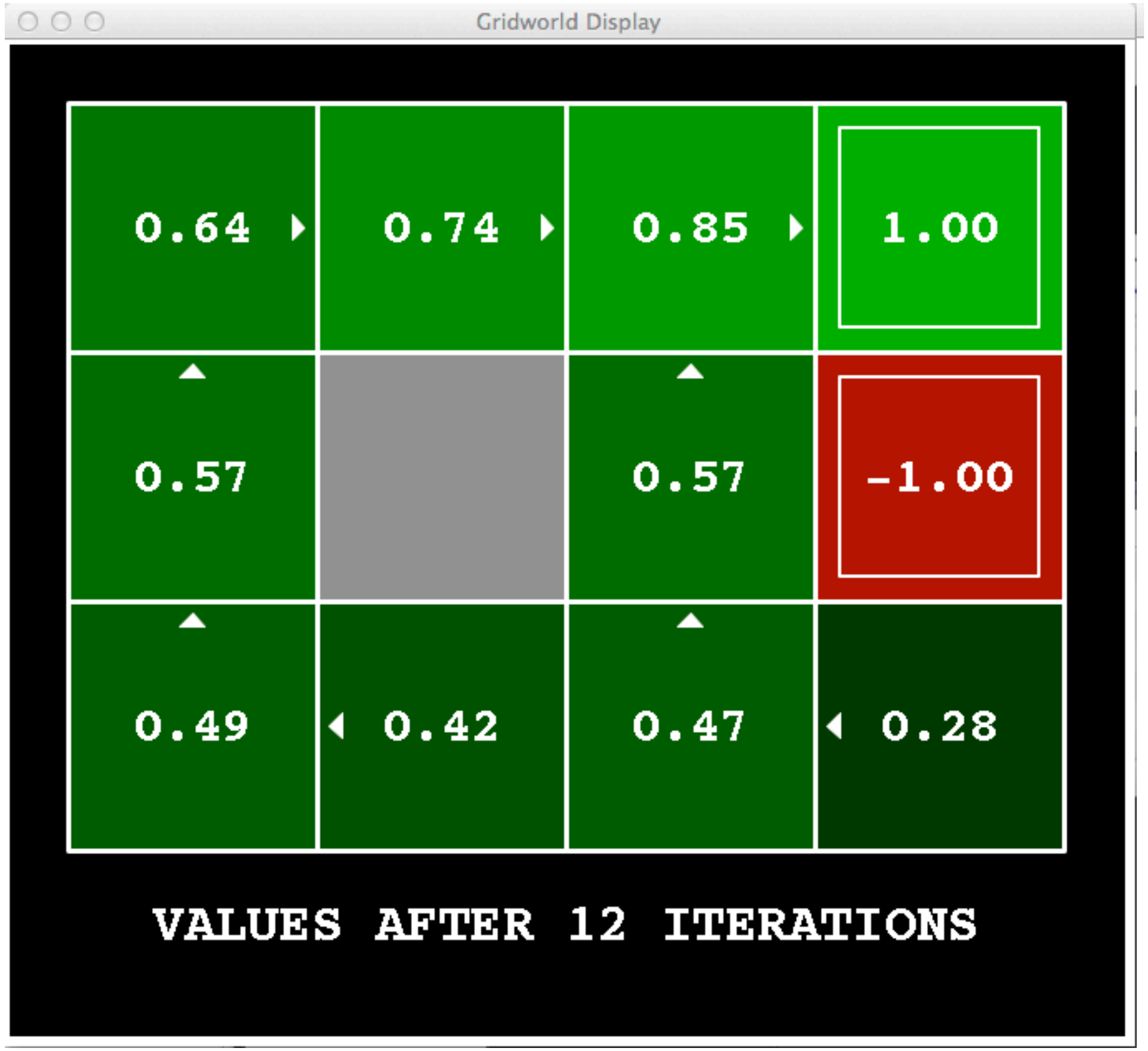
Value iteration example



0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.56		▲ 0.57	-1.00
▲ 0.48	◀ 0.42	▲ 0.47	◀ 0.27

VALUES AFTER 11 ITERATIONS

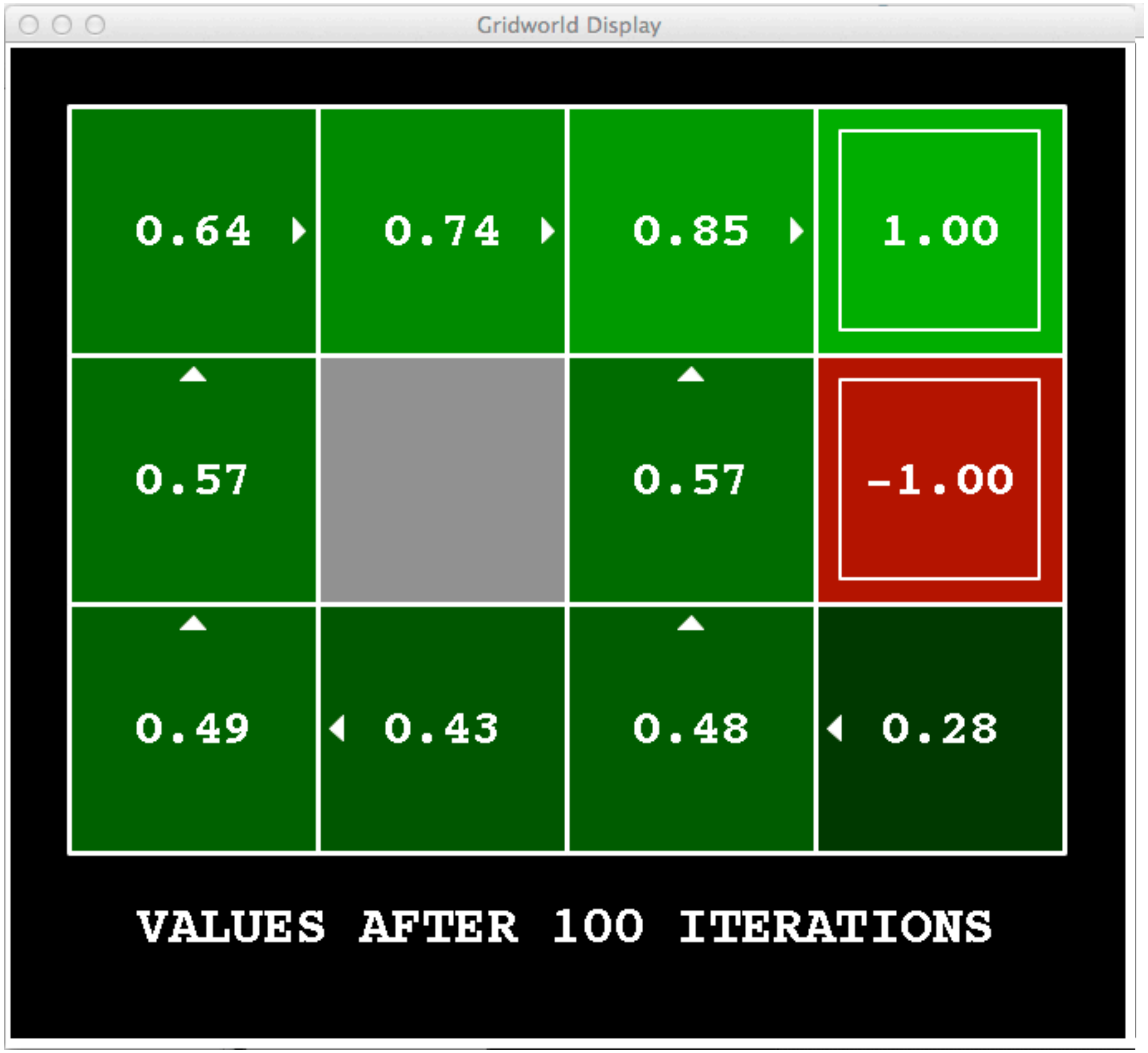
Value iteration example



0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.57		▲ 0.57	-1.00
▲ 0.49	◀ 0.42	▲ 0.47	◀ 0.28

VALUES AFTER 12 ITERATIONS

Value iteration example



0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.57		▴ 0.57	-1.00
▴ 0.49	◂ 0.43	▴ 0.48	◂ 0.28

VALUES AFTER 100 ITERATIONS

Policy iteration

Alternative approach for optimal values:

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

This is policy iteration

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

Policy iteration

Algorithm:

```
$\pi \leftarrow$ an arbitrary initial policy
repeat until no change in $\pi$
    compute utilities given $\pi$
    update $\pi$ as if utilities were correct (i.e., local MEU)
```

To compute utilities given a fixed π (policy evaluation)

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') \quad \text{for all } s$$

i.e., n simultaneous linear equations in n unknowns, solve in $O(n^3)$

Modified policy iteration

Policy iteration often converges in few iterations, but each is expensive

Idea: use a few steps of value iteration (but with π fixed) starting from the value function produced the last time to produce an approximate value determination step.

Often converges much faster than pure VI or PI

Leads to much more general algorithms where Bellman value updates and Howard policy updates can be performed locally in any order

Reinforcement learning algorithms operate by performing such updates based on the observed transitions made in an initially unknown environment

Linear function approximation

What if the state space is large or continuous?

Instead of discretizing the state space (exponential in state variables) define a set of *basis functions* $\phi_1, \ldots, \phi_n$ over X

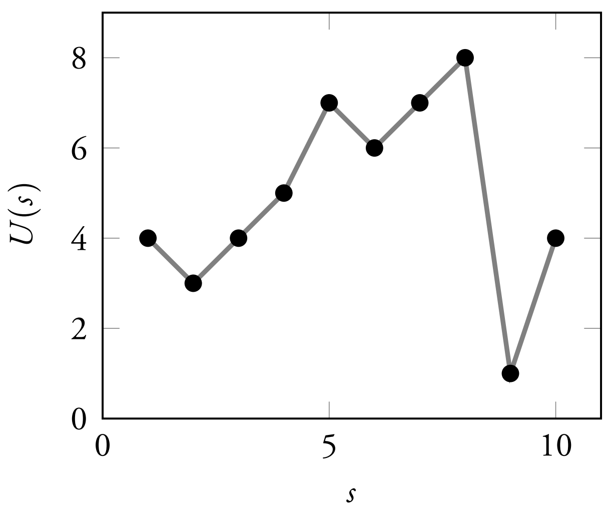
Each basis function maps states in X to real numbers

Think of basis functions as a state-dependent features

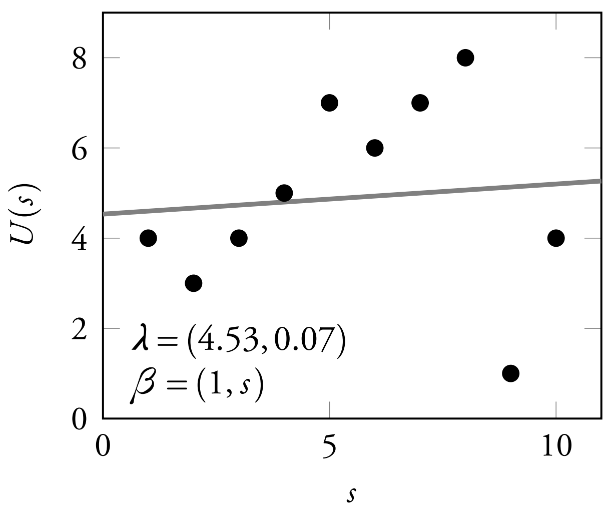
Represent V as a *linear combination* of basis functions: $V(x) = \theta_1\phi_1(x)+ \ldots +\theta_n\phi_n(x)$

Given fixed basis functions, determine $\theta_1, \ldots, \theta_n$ that best represents the optimal value function

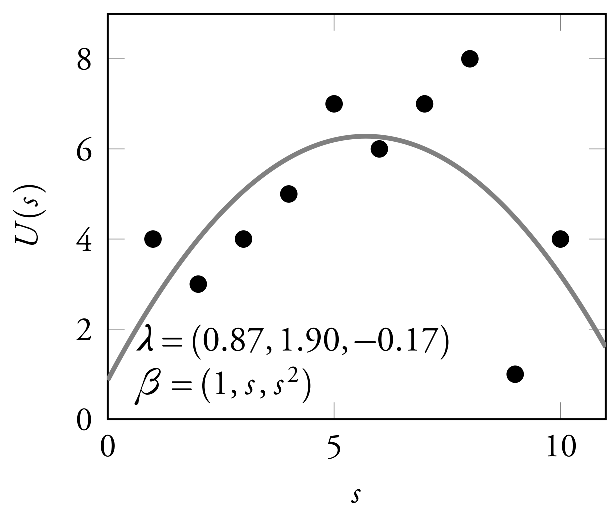
Approximation results



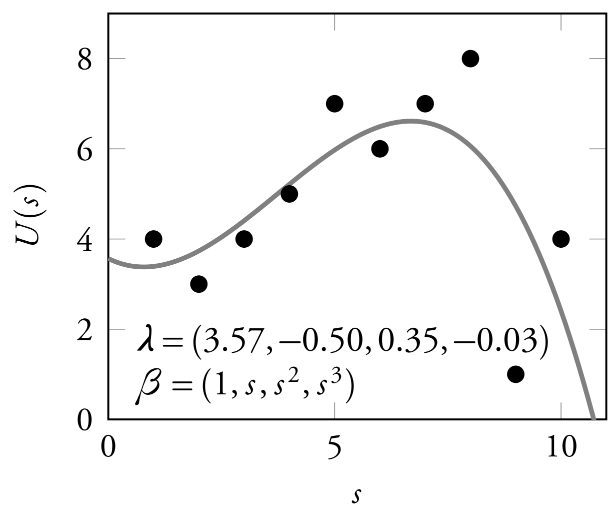
(a) Linear interpolation.



(b) Linear regression (linear basis).



(c) Linear regression (quadratic basis).



(d) Linear regression (cubic basis).

Online methods

Solving for a full policy offline is expensive!

What can we do?

Online methods

Online methods compute optimal action from current state

Expand tree up to some horizon

States reachable from the current state is typically small compared to full state space

Heuristics and branch-and-bound techniques allow search space to be pruned

Monte Carlo methods provide approximate solutions

Forward search

Provides optimal action from current state s up to depth d

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

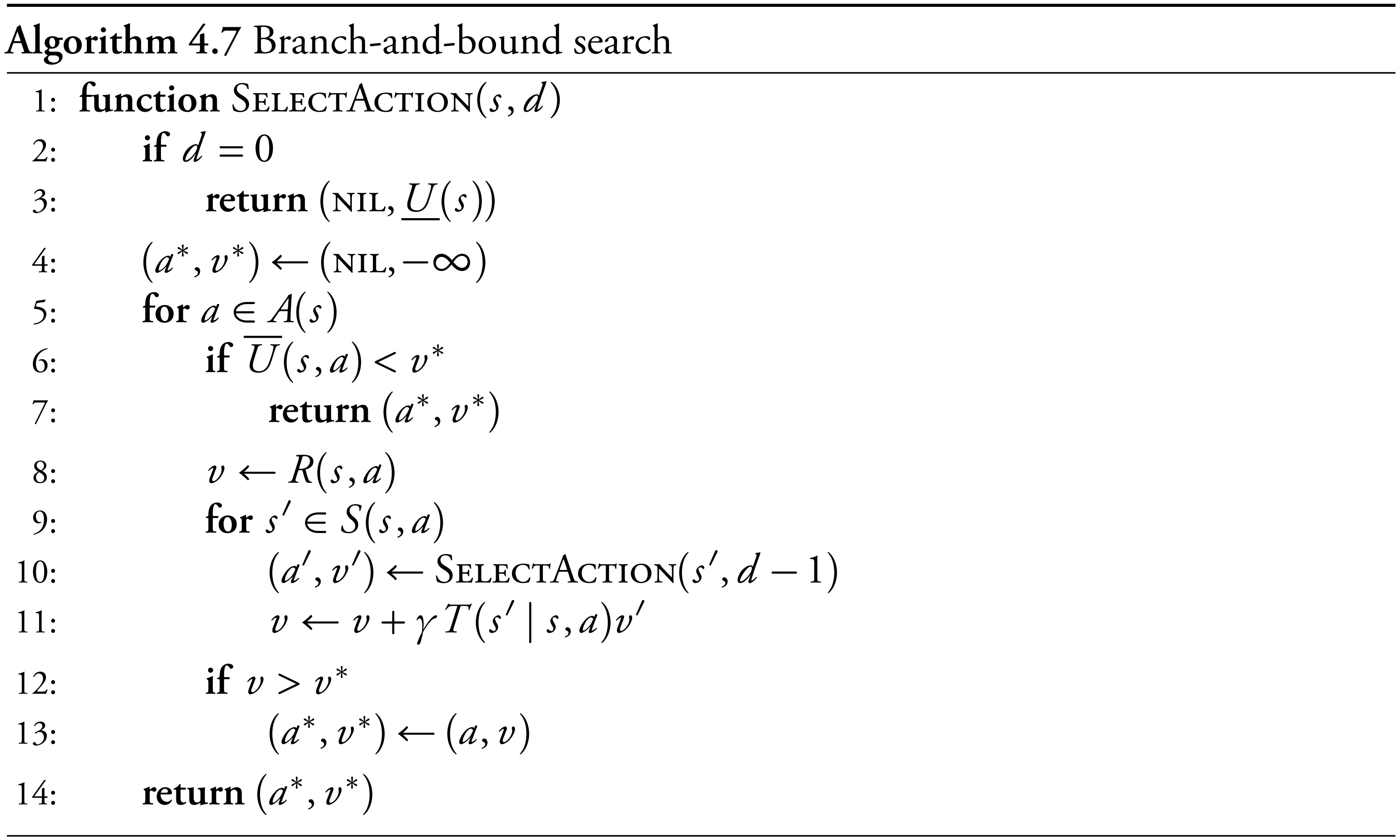
Algorithm 4.6 Forward search

```
1: function SelectAction($s, d$)
2:     if $d = 0$
3:         return (NIL, 0)
4:     $(a^*, v^*) \leftarrow$ (NIL, $-\infty$)
5:     for $a \in A(s)$
6:         $v \leftarrow R(s, a)$
7:         for $s' \in S(s, a)$
8:             $(a', v') \leftarrow$ SelectAction($s', d - 1$)
9:             $v \leftarrow v + \gamma T(s' \mid s, a) v'$
10:        if $v > v^*$
11:            $(a^*, v^*) \leftarrow (a, v)$
12:    return $(a^*, v^*)$
```

Time complexity is $O((|S| \text{ x } |A|)^d)$

Branch and bound search

Requires a lower bound $\underline{U}(s)$ and upper bound $\bar{U}(s)$



Algorithm 4.7 Branch-and-bound search

```
function SelectAction($s, d$)
    if $d = 0$
        return (nil, $\underline{U}(s)$)
    $(a^*, v^*) \leftarrow$ (nil, $-\infty$)
    for $a \in A(s)$
        if $\overline{U}(s, a) < v^*$
            return $(a^*, v^*)$
        $v \leftarrow R(s, a)$
        for $s' \in S(s, a)$
            $(a', v') \leftarrow$ SelectAction($s', d - 1$)
            $v \leftarrow v + \gamma T(s' \mid s, a) v'$
        if $v > v^*$
            $(a^*, v^*) \leftarrow (a, v)$
    return $(a^*, v^*)$
```

Worse case complexity?

Monte Carlo evaluation

Algorithm 4.11 Monte Carlo policy evaluation

```
1: function MonteCarloPolicyEvaluation($\lambda, d$)
2:     for $i \leftarrow 1$ to $n$
3:         $s \sim b$
4:         $u_i \leftarrow$ Rollout($s, d, \pi_\lambda$)
5:     return $\frac{1}{n}\sum_{i=1}^{n} u_i$
```

Algorithm 4.10 Rollout evaluation

```
1: function Rollout($s, d, \pi_0$)
2:     if $d = 0$
3:         return 0
4:     $a \sim \pi_0(s)$
5:     $(s', r) \sim G(s, a)$
6:     return $r + \gamma$Rollout($s', d - 1, \pi_0$)
```

Estimate value of a policy by sampling from a simulator

Sparse sampling

Requires a generative model $(s',r) \sim G(s,a)$

Algorithm 4.8 Sparse sampling

```
function SelectAction($s, d$)
    if $d = 0$
        return $(\text{NIL}, 0)$
    $(a^*, v^*) \leftarrow (\text{NIL}, -\infty)$
    for $a \in A(s)$
        $v \leftarrow 0$
        for $i \leftarrow 1$ to $n$
            $(s', r) \sim G(s, a)$
            $(a', v') \leftarrow$ SelectAction$(s', d-1)$
            $v \leftarrow v + (r + \gamma v')/n$
        if $v > v^*$
            $(a^*, v^*) \leftarrow (a, v)$
    return $(a^*, v^*)$
```

Complexity? Guarantees?

Sparse sampling

Requires a generative model $(s', r) \sim G(s, a)$

Algorithm 4.8 Sparse sampling

```
function SelectAction($s, d$)
    if $d = 0$
        return $(\text{NIL}, 0)$
    $(a^*, v^*) \leftarrow (\text{NIL}, -\infty)$
    for $a \in A(s)$
        $v \leftarrow 0$
        for $i \leftarrow 1$ to $n$
            $(s', r) \sim G(s, a)$
            $(a', v') \leftarrow$ SelectAction($s', d - 1$)
            $v \leftarrow v + (r + \gamma v')/n$
        if $v > v^*$
            $(a^*, v^*) \leftarrow (a, v)$
    return $(a^*, v^*)$
```

Complexity = $O((n \times |A|)^d)$, Guarantees = probabilistic

Monte Carlo tree search

Algorithm 4.9 Monte Carlo tree search

```
1:  function SelectAction($s, d$)
2:      loop
3:          Simulate($s, d, \pi_0$)
4:      return $\arg\max_a Q(s, a)$
5:  function Simulate($s, d, \pi_0$)
6:      if $d = 0$
7:          return 0
8:      if $s \notin T$
9:          for $a \in A(s)$
10:             $(N(s, a), Q(s, a)) \leftarrow (N_0(s, a), Q_0(s, a))$
11:         $T = T \cup \{s\}$
12:         return Rollout($s, d, \pi_0$)
13:     $a \leftarrow \arg\max_a Q(s, a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$
14:     $(s', r) \sim G(s, a)$
15:     $q \leftarrow r + \gamma$Simulate($s', d - 1, \pi_0$)
16:     $N(s, a) \leftarrow N(s, a) + 1$
17:     $Q(s, a) \leftarrow Q(s, a) + \frac{q - Q(s,a)}{N(s,a)}$
18:     return $q$
```

UCT (Upper Confident bounds for Trees)

UCT continued

Search (within the tree, T)

- Execute action that maximizes $Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$
- Update the value $Q(s,a)$ and counts $N(s)$ and $N(s,a)$
- c is a exploration constant

Expansion (outside of the tree, T)

- Create a new node for the state
- Initialize $Q(s,a)$ and $N(s,a)$ (usually to 0) for each action

Rollout (outside of the tree, T)

- Only expand once and then use a rollout policy to select actions (e.g., random policy)
- Add the rewards gained during the rollout with those in the tree:

$$r + \gamma \textsc{Rollout}(s', d-1, \pi_0)$$

UCT continued

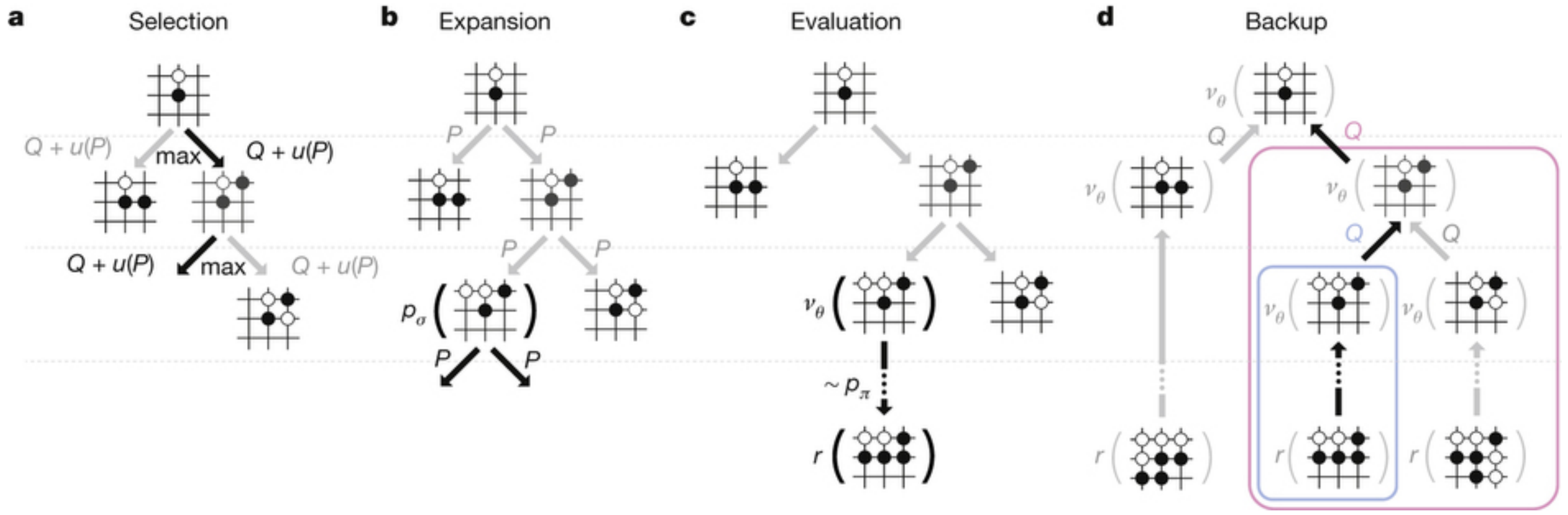
Continue UCT until some termination condition (usually a fixed number of samples)

Complexity?

Guarantees?

AlphaGo

Uses UCT with neural net to approximate opponent choices and state values


a
Selection
Q + u(P)
max
Q + u(P)
Q + u(P)
max
Q + u(P)
b
Expansion
P
P
P
P
p_σ
P
P
c
Evaluation
v_θ
$\sim p_\pi$
r
d
Backup
v_θ
Q
Q
v_θ
v_θ
Q
Q
v_θ
v_θ
r
r
r