Markov Decision Processes

Robert Platt
Northeastern University

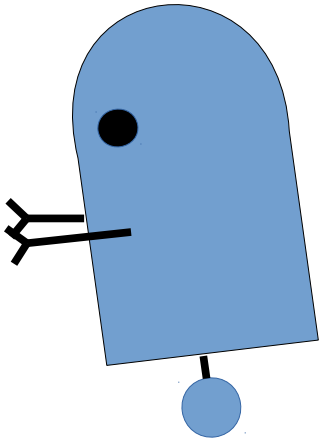
Some images and slides are used from:
1. CS188 UC Berkeley
2. AIMA
3. Chris Amato

Stochastic domains

So far, we have studied search

Can use search to solve simple planning problems,
e.g. robot planning using A*

But only in deterministic domains...

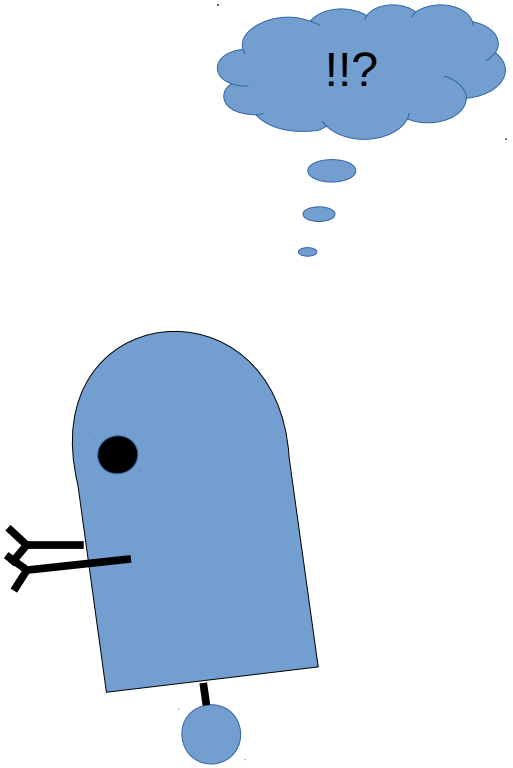


Stochastic domains

So far, we have studied search

Can use search to solve simple planning problems,
e.g. robot planning using A*

A* doesn't work so well in stochastic environments...

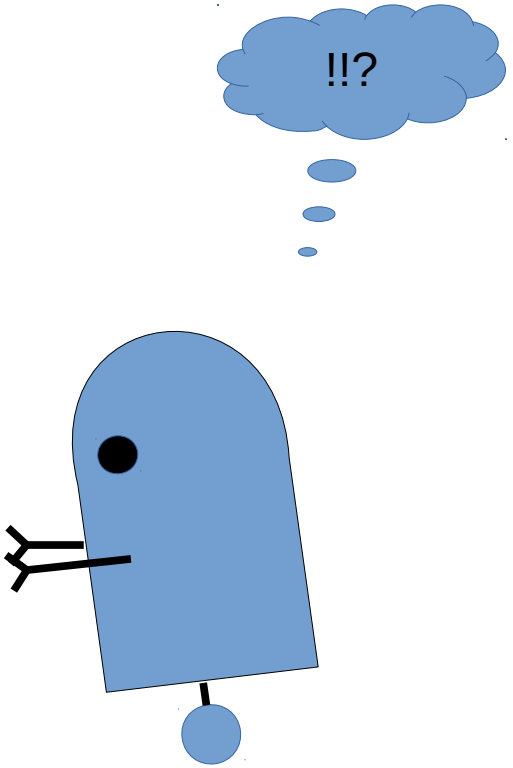


Stochastic domains

So far, we have studied search

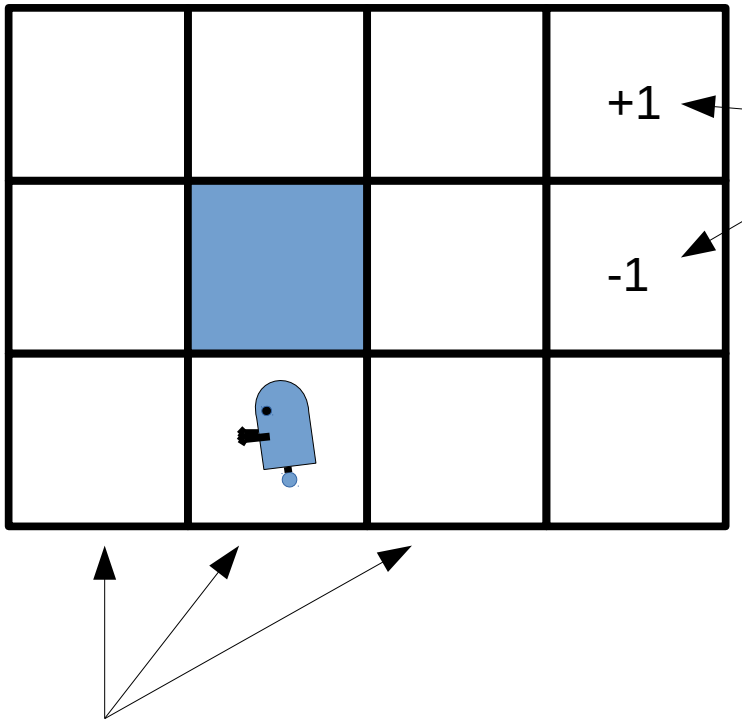
Can use search to solve simple planning problems,
e.g. robot planning using A*

A* doesn't work so well in stochastic environments...



We are going to introduce a new framework for encoding problems
w/ stochastic dynamics: the Markov Decision Process (MDP)

Markov Decision Process (MDP): grid world example



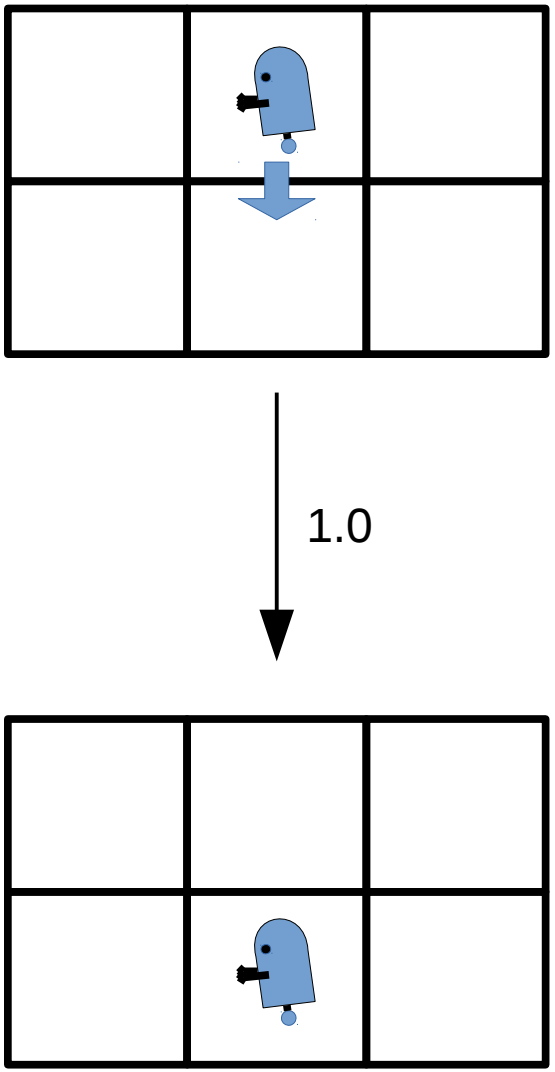
Rewards:
– agent gets these rewards in these cells
– goal of agent is to maximize reward

Actions: left, right, up, down
– take one action per time step
– actions are stochastic: only go in intended direction 80% of the time

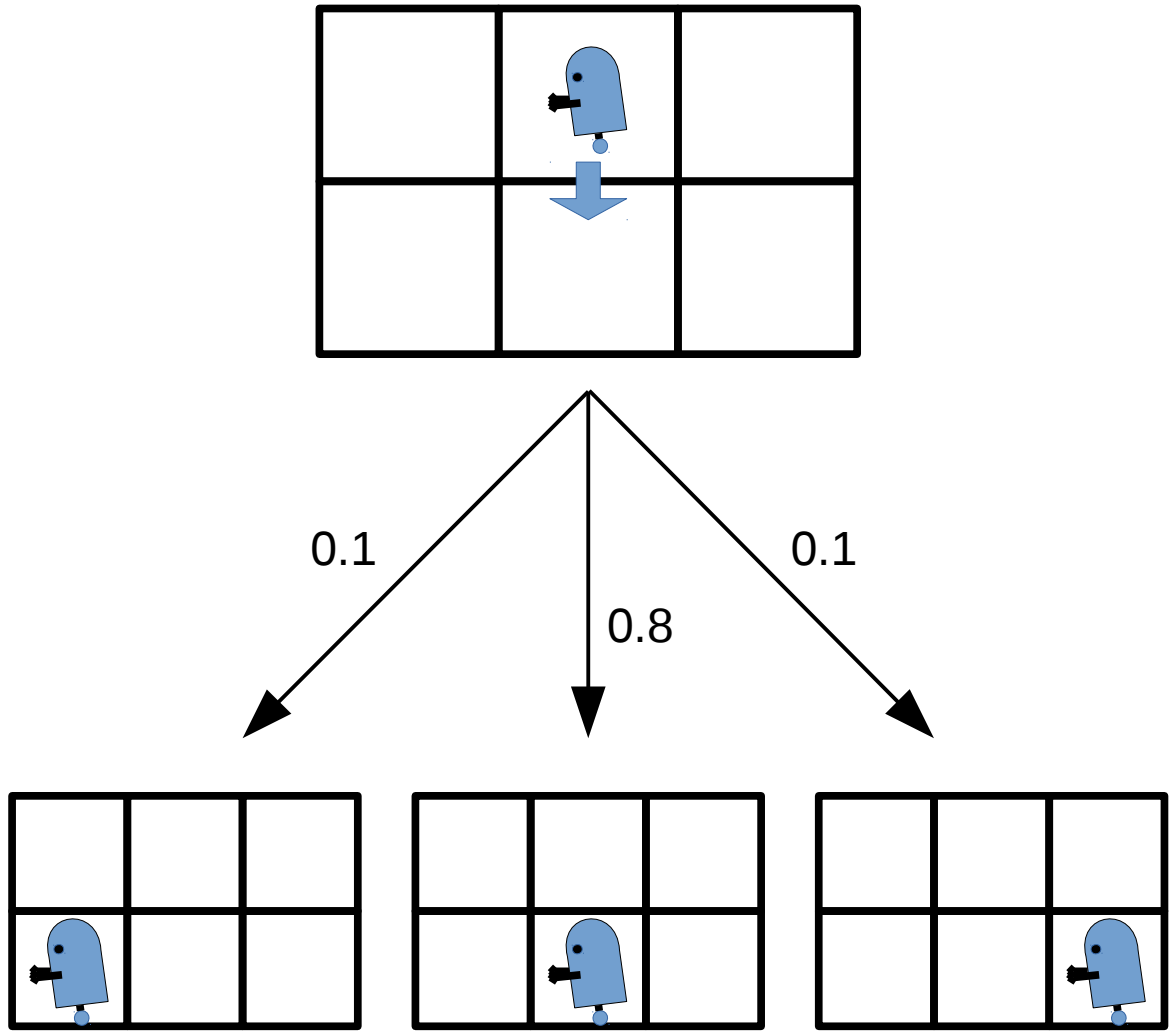
States:
– each cell is a state

Markov Decision Process (MDP)

Deterministic
– same action always has same outcome

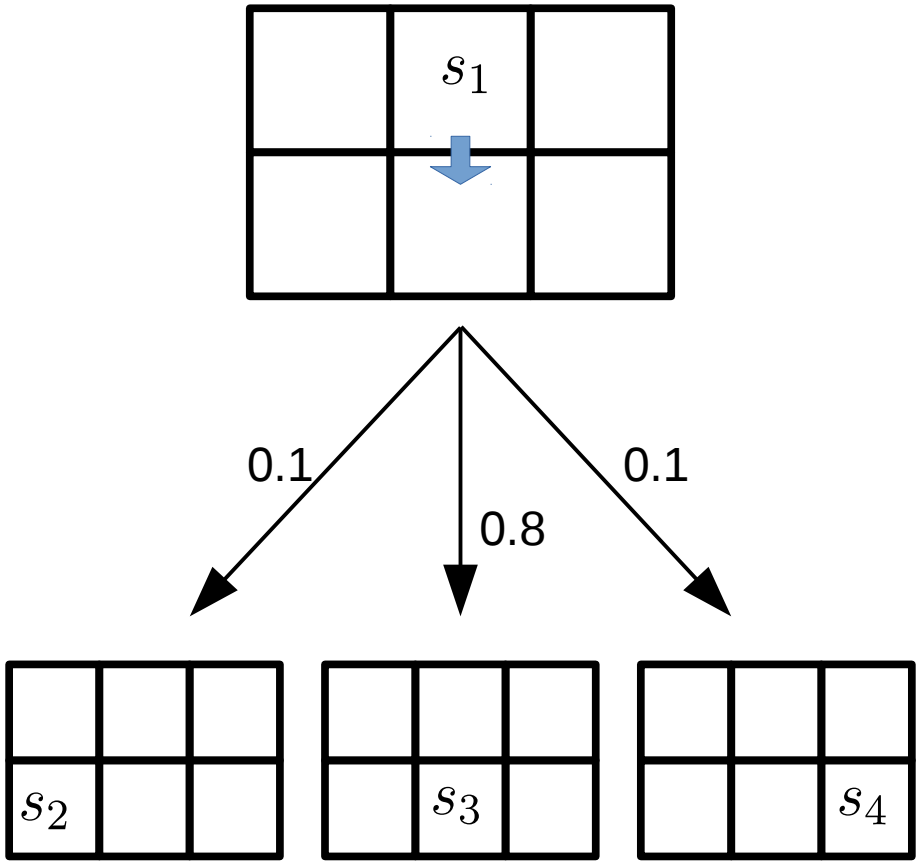


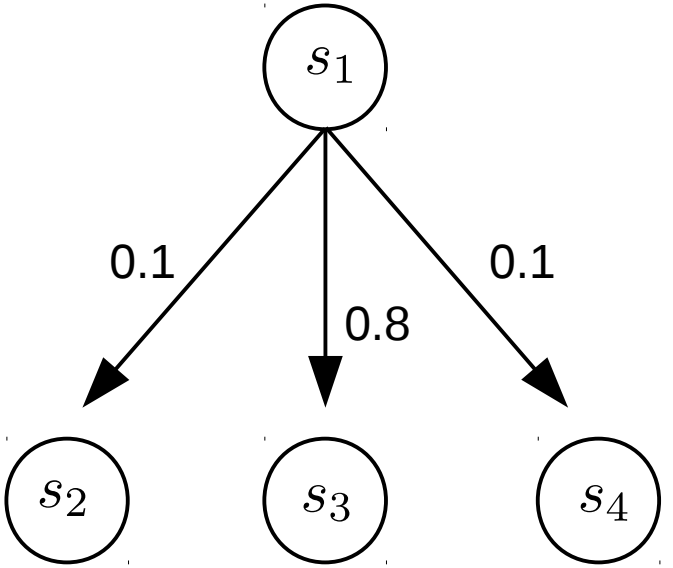
Stochastic
– same action could have different outcomes



Markov Decision Process (MDP)

Same action could have different outcomes:



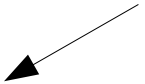


Transition function at s_1:

s'	T(s,a,s')
s_2	0.1
s_3	0.8
s_4	0.1

Markov Decision Process (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: $R : S \times A \to \mathbb{R}_{\geq 0}$

Markov Decision Process (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: $R : S \times A \to \mathbb{R}_{\geq 0}$

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

Markov Decision Process (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: $R : S \times A \to \mathbb{R}_{\geq 0}$

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

But, what is the objective?

Markov Decision Process (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: $R : S \times A \to \mathbb{R}_{\geq 0}$

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

Objective: calculate a strategy for acting so as to maximize the future rewards.

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

What is a policy?

A policy tells the agent what action to execute as a function of state:

Deterministic policy: $\pi(s) : S \rightarrow A$

– agent always executes the same action from a given state

Stochastic policy: $\pi(s, a) : S \times A \rightarrow \mathbb{R}$

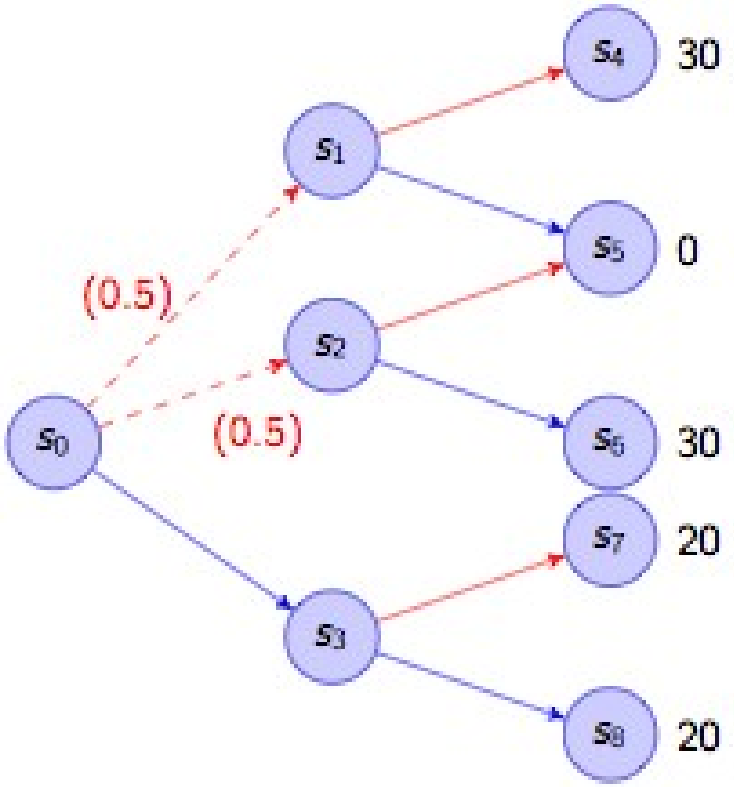
– agent selects an action to execute by drawing from a *probability distribution* encoded by the policy *...*

Policies versus Plans

Policies are more general than *plans*

Plan:
– specifies a sequence of actions to execute
– cannot react to unexpected outcome

Policy:
– tells you what action to take from *any* state



Plan might not be optimal

U(r,r)=15

U(r,b)=15

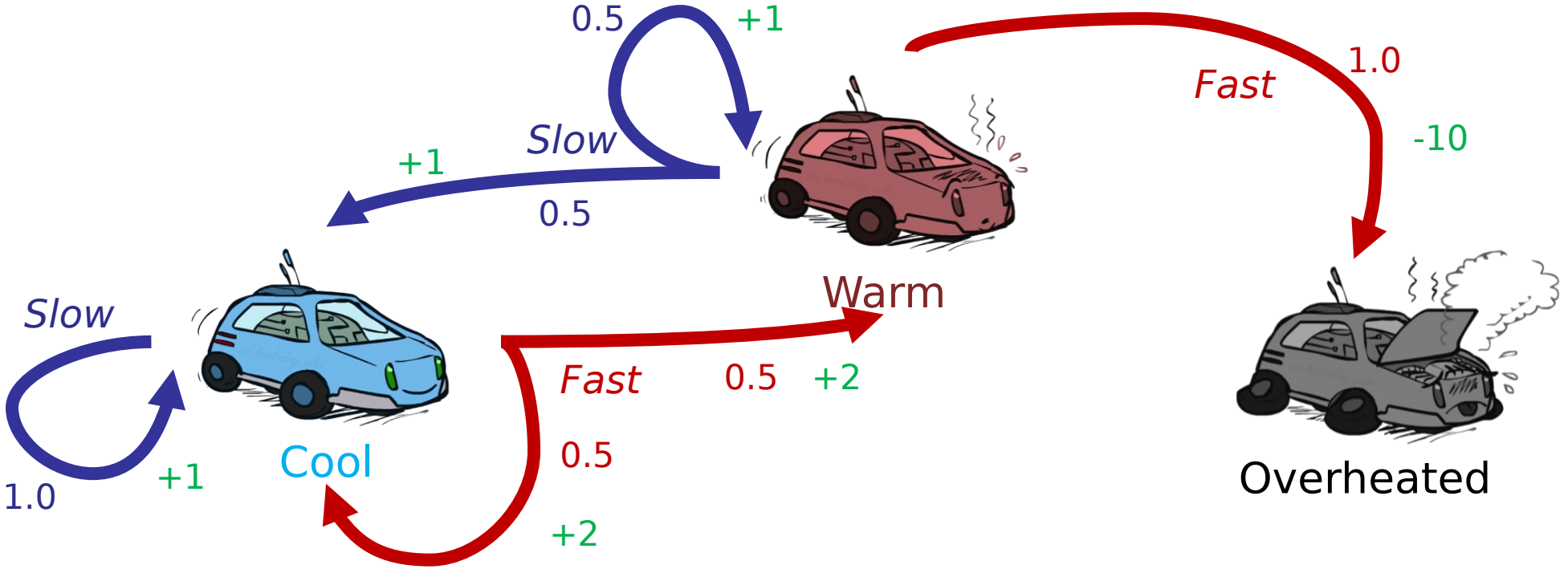
U(b,r)=20

U(b,b)=20

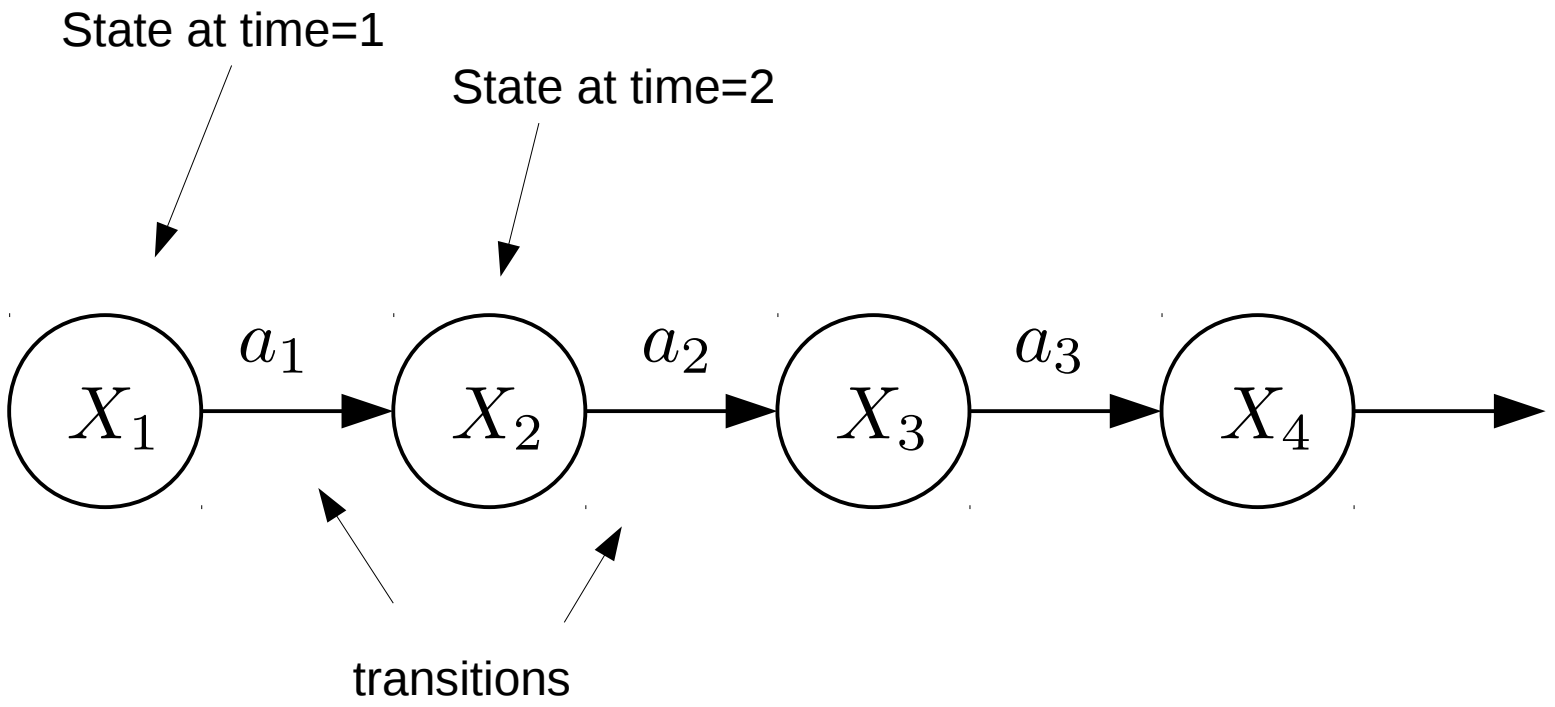
The optimal policy can achieve U=30

Another example of an MDP

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



Markov?



Since this is a Markov process, we assume transitions are Markov:

Transition dynamics: $P(X_t|a_{t-1}, X_{t-1}) = P(X_t|a_{t-1}, X_{t-1}, \ldots, X_1)$

Markov assumption: $X_t \perp\!\!\!\perp X_{t-2} | X_{t-1}$

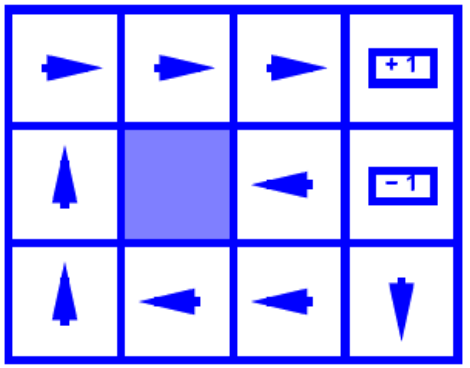
Conditional independence

Objective: maximize expected future reward

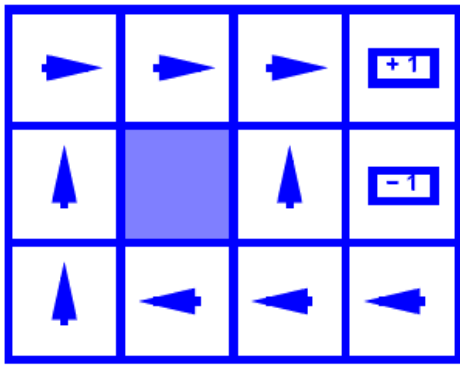
$$R_t \equiv \quad \text{Expected future reward starting at time t}$$

$$= \mathbb{E} \sum_{i=0}^{\infty} r_{t+i}$$

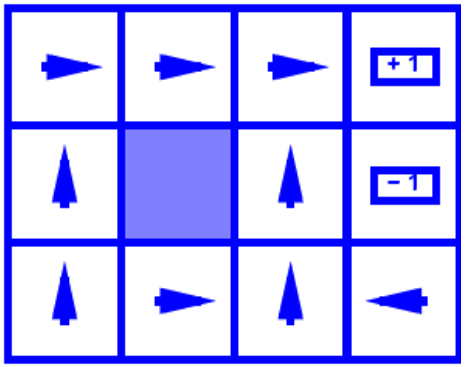
Examples of optimal policies



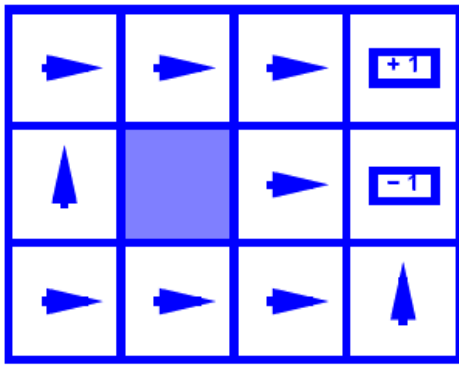
R(s) = -0.01



R(s) = -0.03



R(s) = -0.4



R(s) = -2.0

Objective: maximize expected future reward

$R_t \equiv$ Expected future reward starting at time t

$$= \mathbb{E} \sum_{i=0}^{\infty} r_{t+i}$$

What's wrong w/ this?

Objective: maximize expected future reward

$R_t \equiv$ Expected future reward starting at time t

$$= \mathbb{E} \sum_{i=0}^{\infty} r_{t+i}$$

What's wrong w/ this?

<u>Two viable alternatives:</u>

1. maximize expected future reward *over the next T timesteps* (finite horizon):

$$R_t = \mathbb{E} \sum_{i=0}^{T} r_{t+i}$$

2. maximize expected *discounted* future rewards:

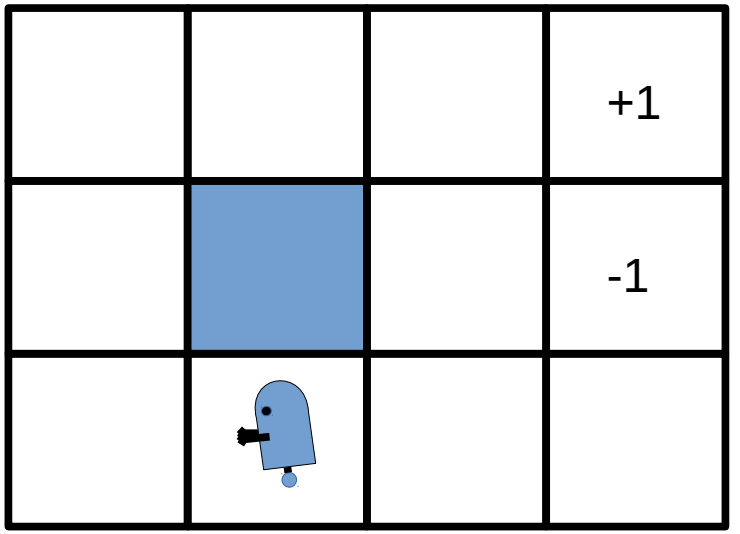
$$R_t = \mathbb{E} \sum_{i=0}^{\infty} \gamma^i r_{t+i} = \mathbb{E} \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \right]$$

Discount factor (usually around 0.9): $0 < \gamma < 1$

Choosing a reward function

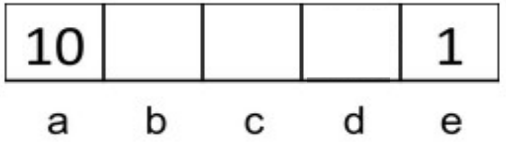
A few possibilities:
– all reward on goal
– negative reward everywhere
  except terminal states
– gradually increasing reward
  as you approach the goal

In general:
– reward can be whatever you
  want



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?

Value functions

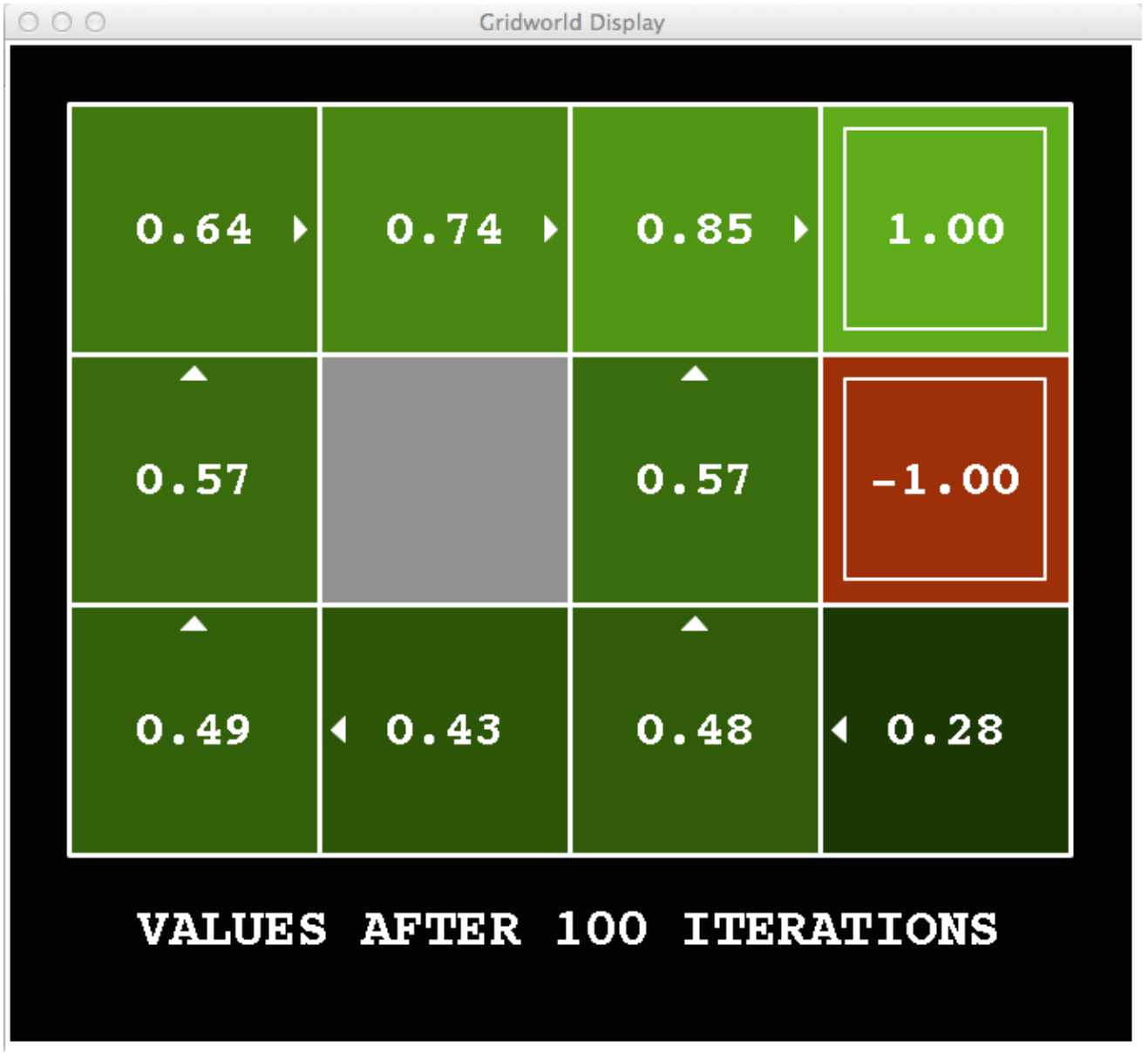
$V(s) \equiv$ Expected discounted reward if agent acts optimally starting in state s (value function).

<u>Game plan:</u>

1. calculate the optimal value function

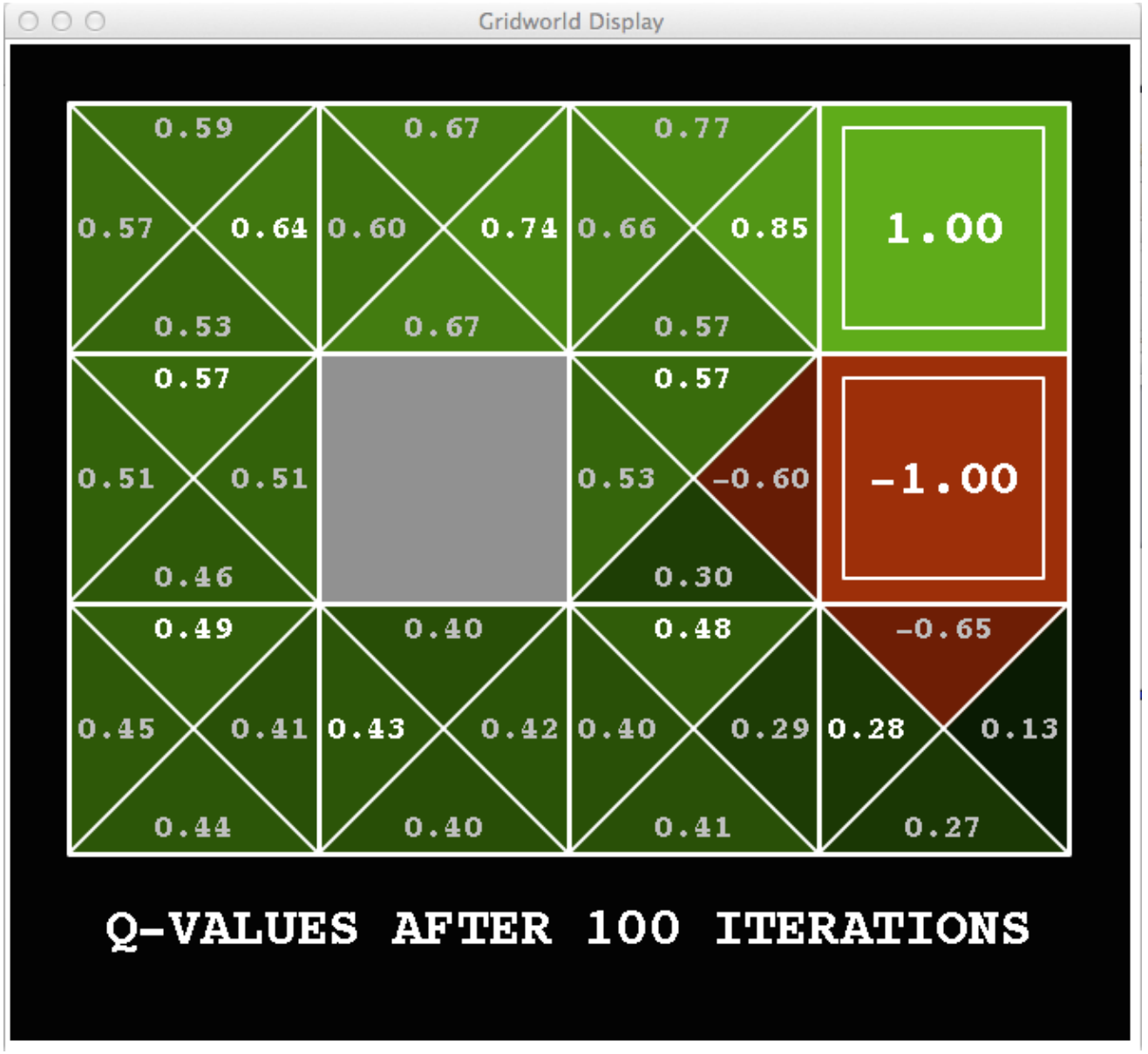
2. calculate optimal policy from optimal value function

Grid world optimal value function



Noise = 0.2
Discount = 0.9
Living reward = 0

Grid world optimal action-value function



Noise = 0.2
Discount = 0.9
Living reward = 0

Value iteration

How do we calculate the optimal value function?

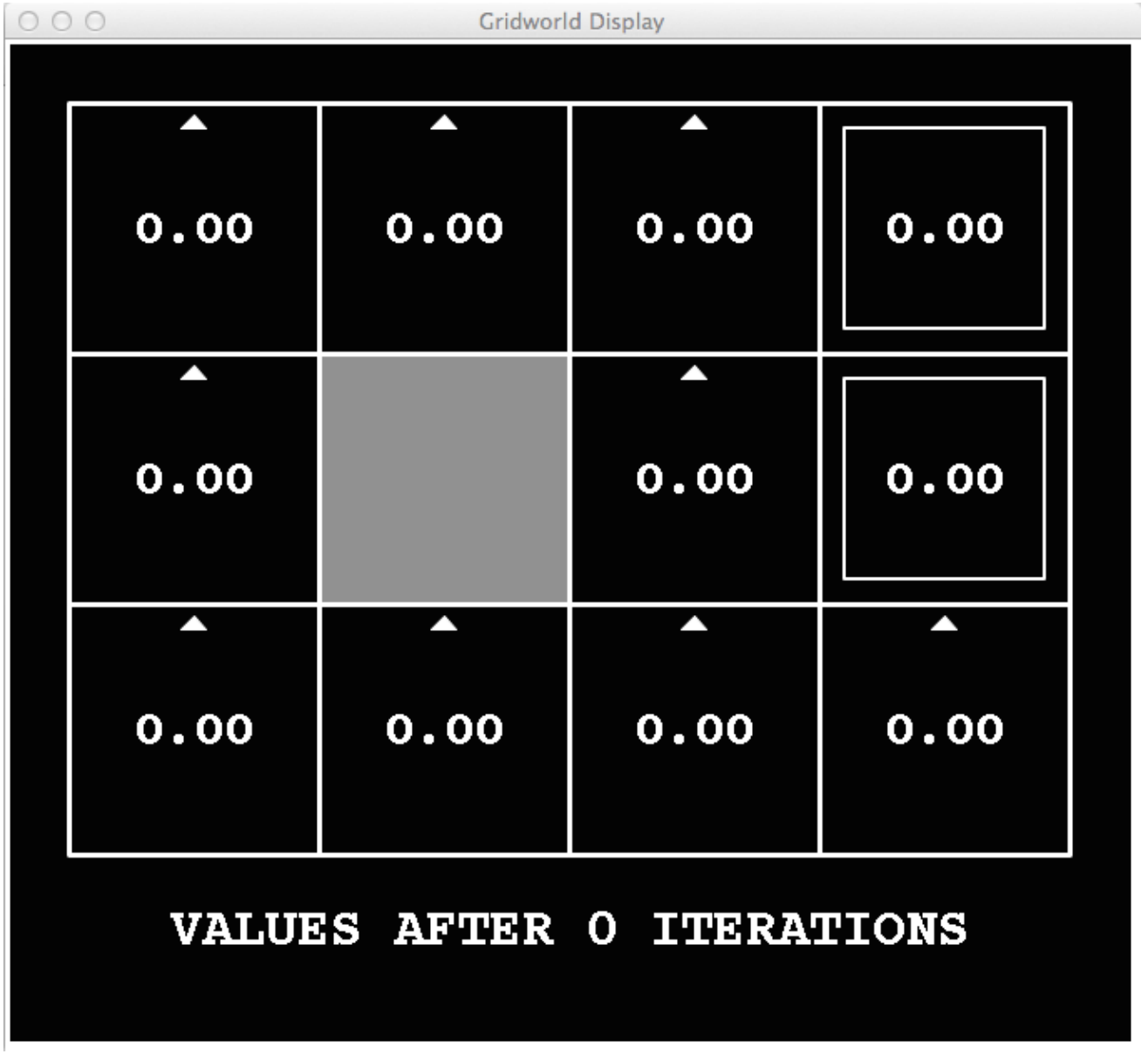
Answer: Value Iteration!

Value Iteration
Input: MDP=(S,A,T,r)
Output: value function, V

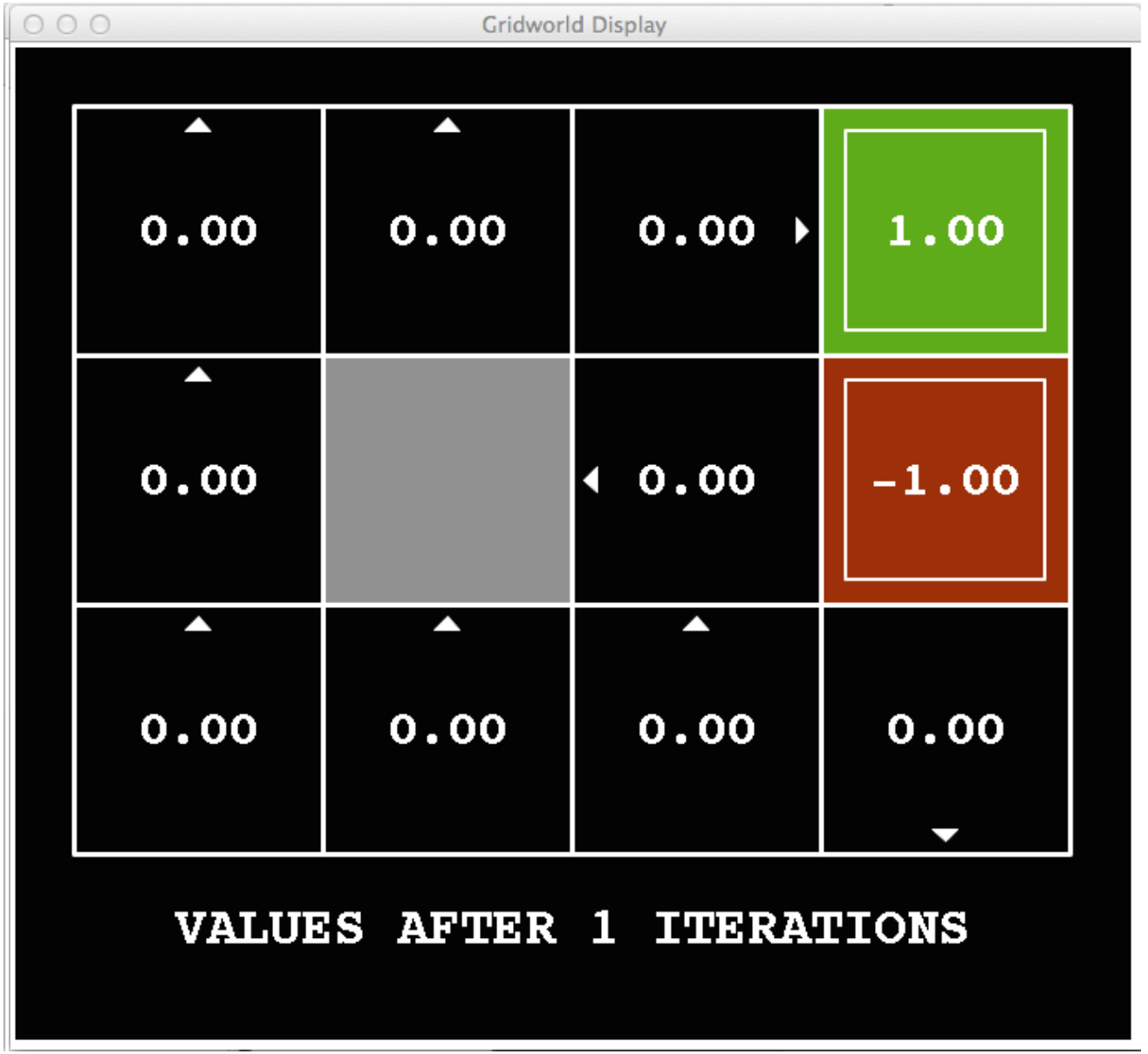
1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $$V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[r(s,a) + \gamma V_i(s') \right]$$
5. if V converged, then break

Value iteration example

Noise = 0.2
Discount = 0.9
Living reward = 0



Value iteration example



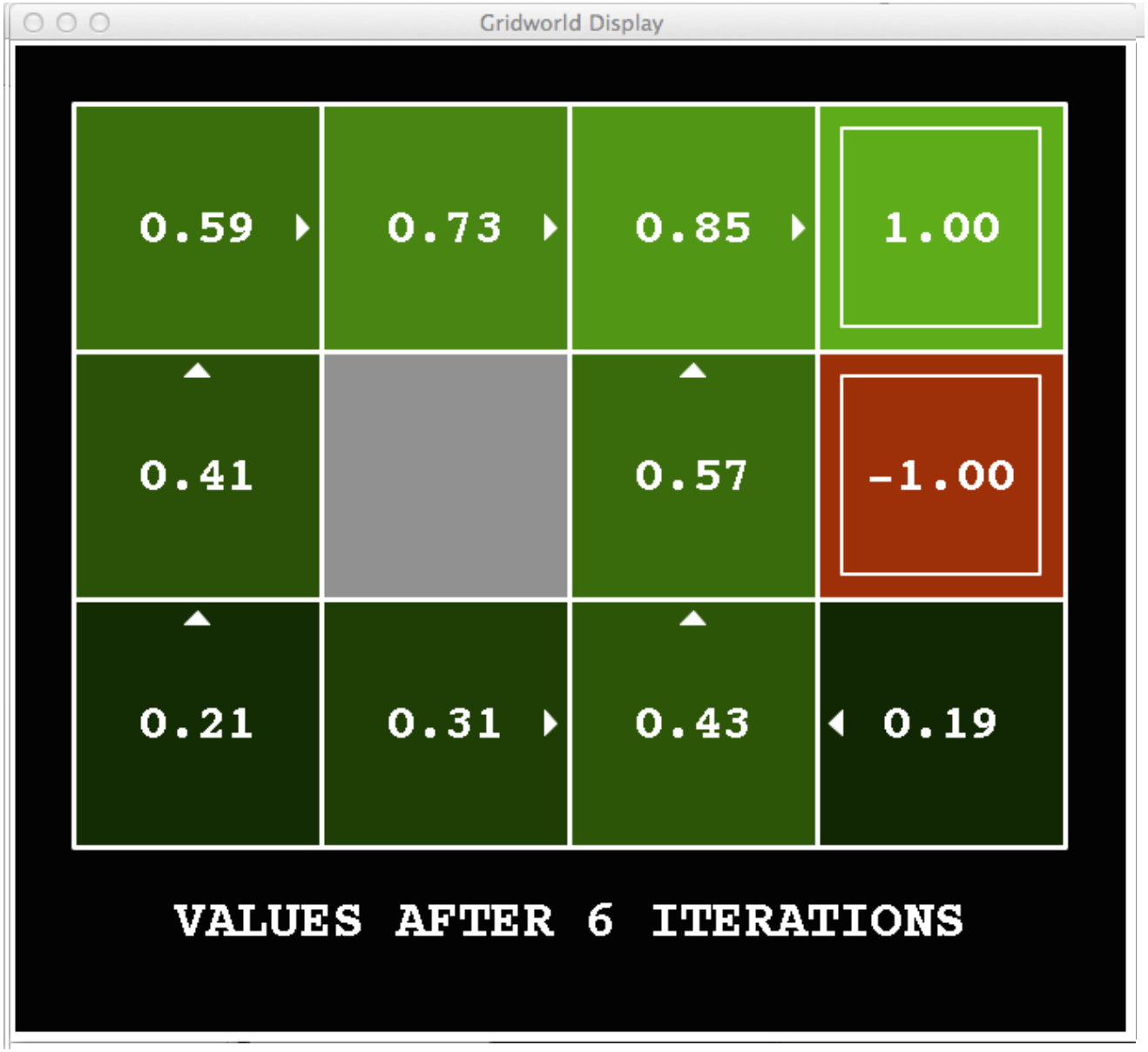
Value iteration example

Value iteration example

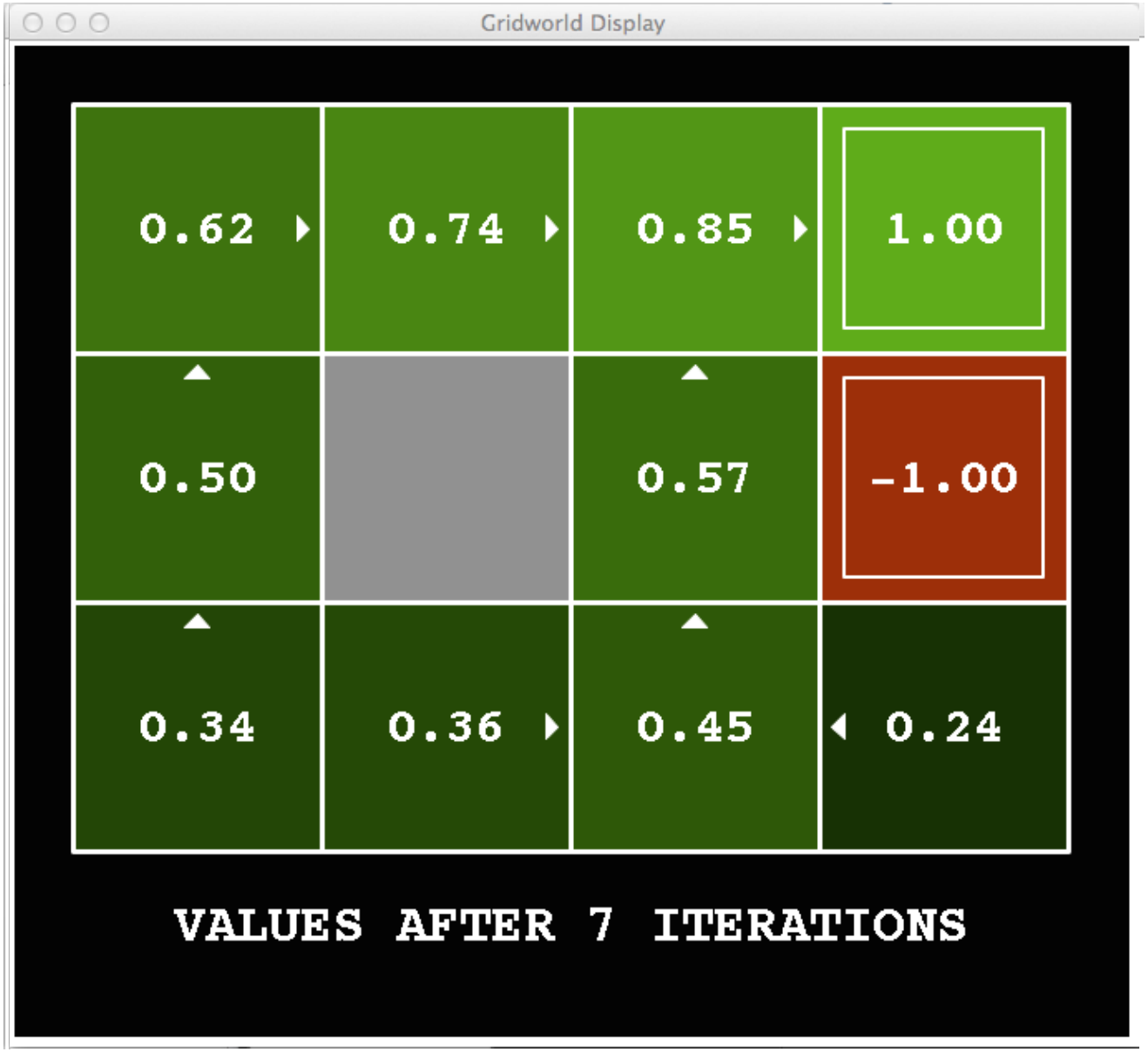
Value iteration example

Value iteration example

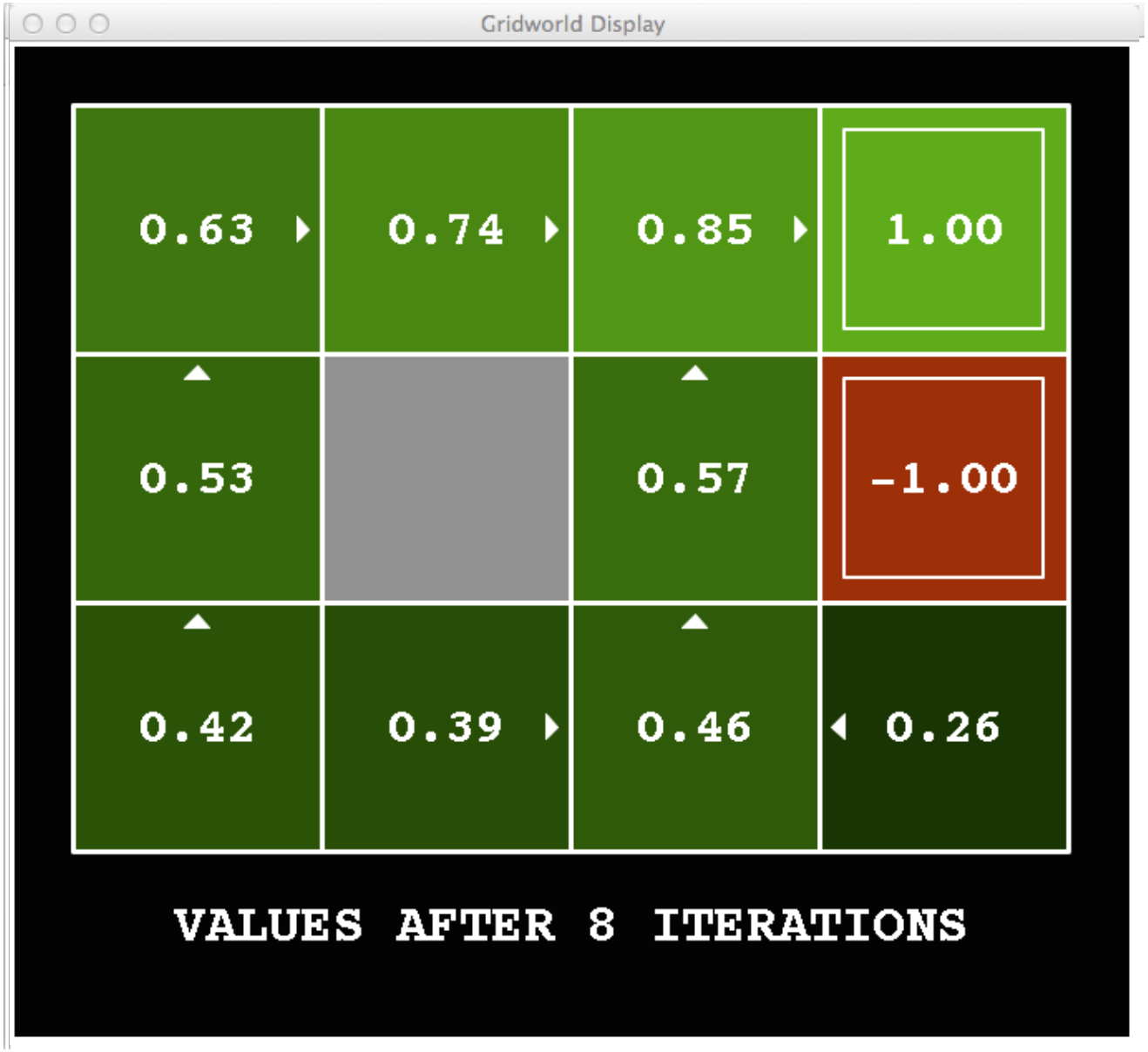
Value iteration example



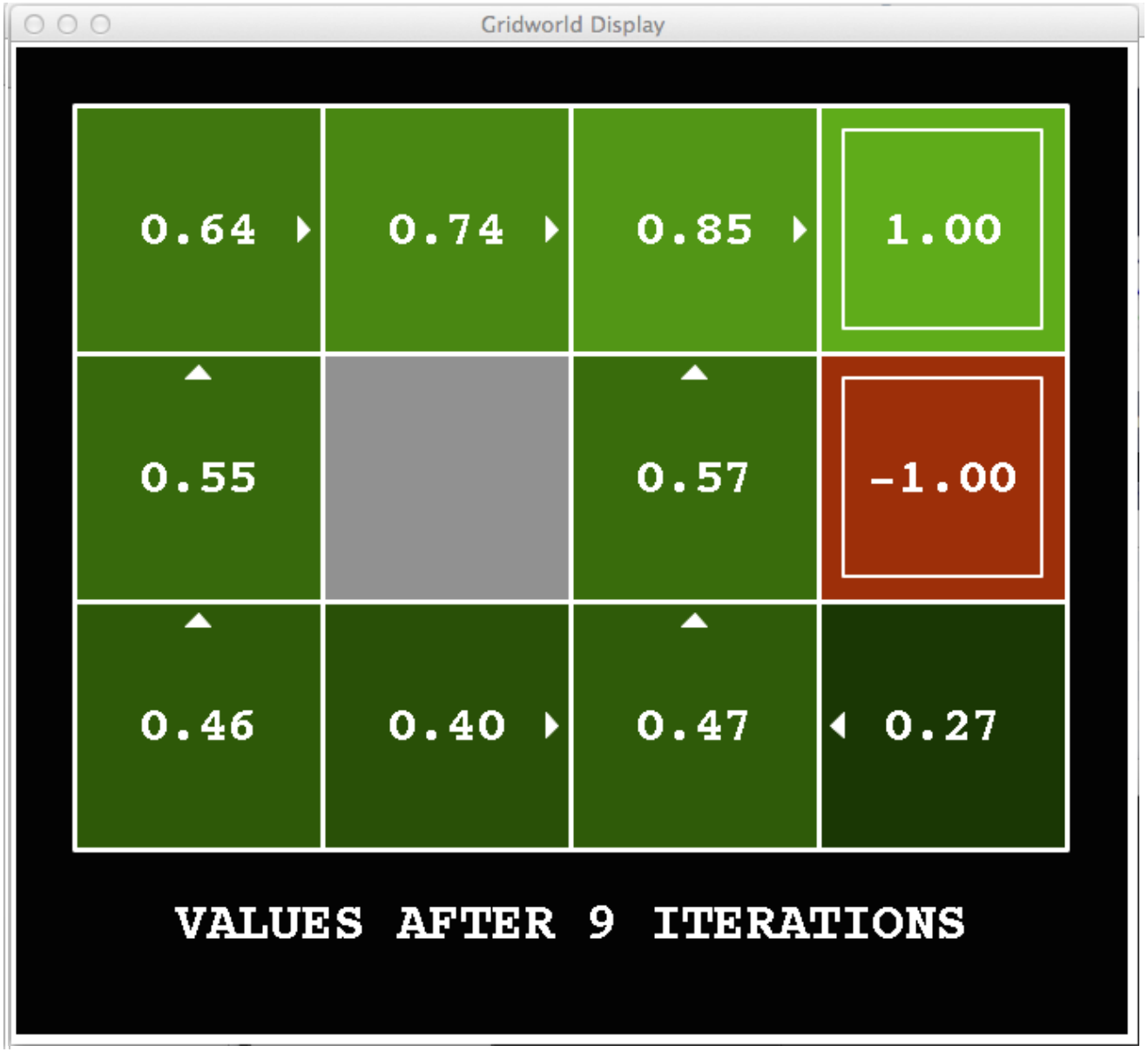
Value iteration example



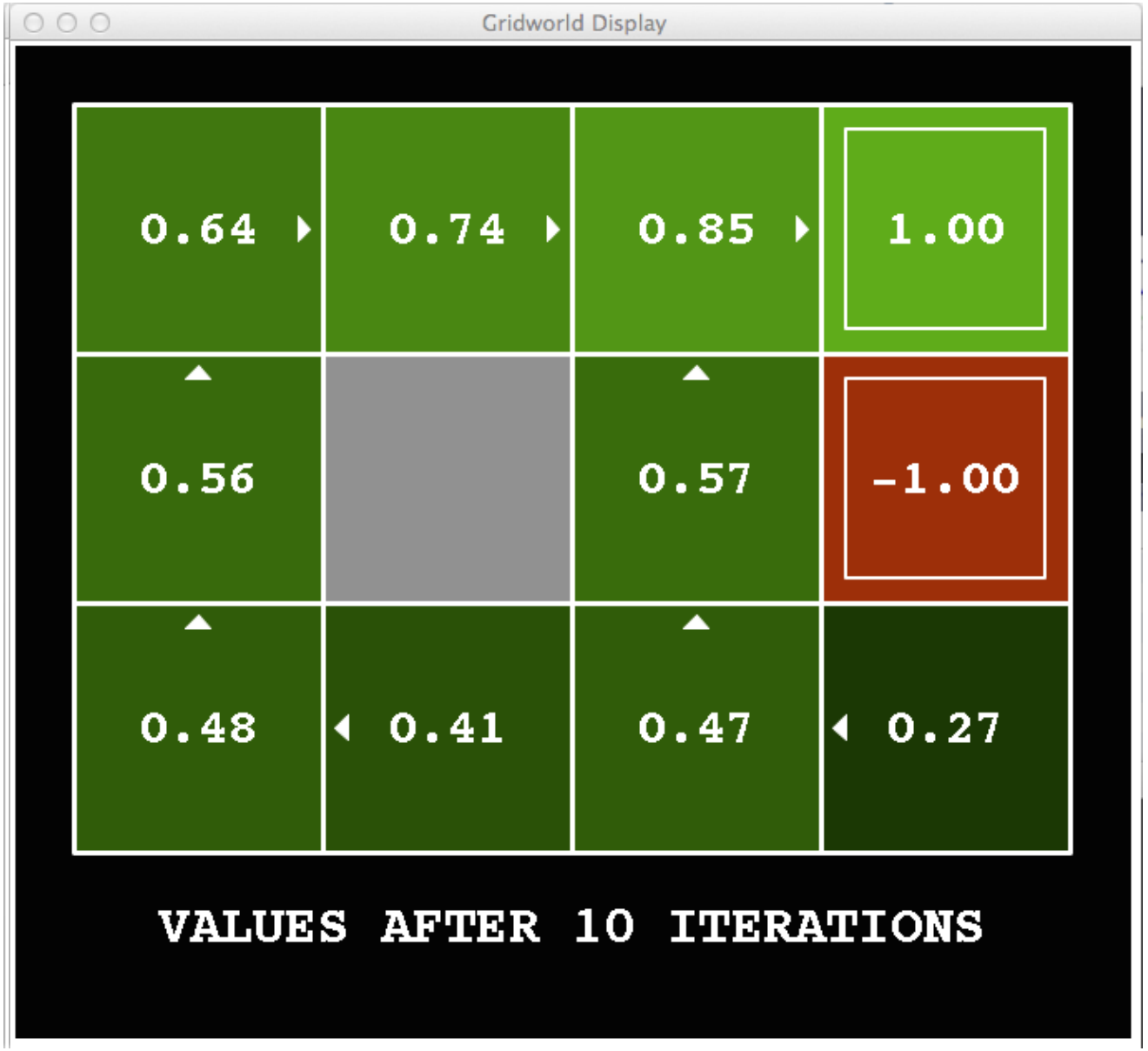
Value iteration example



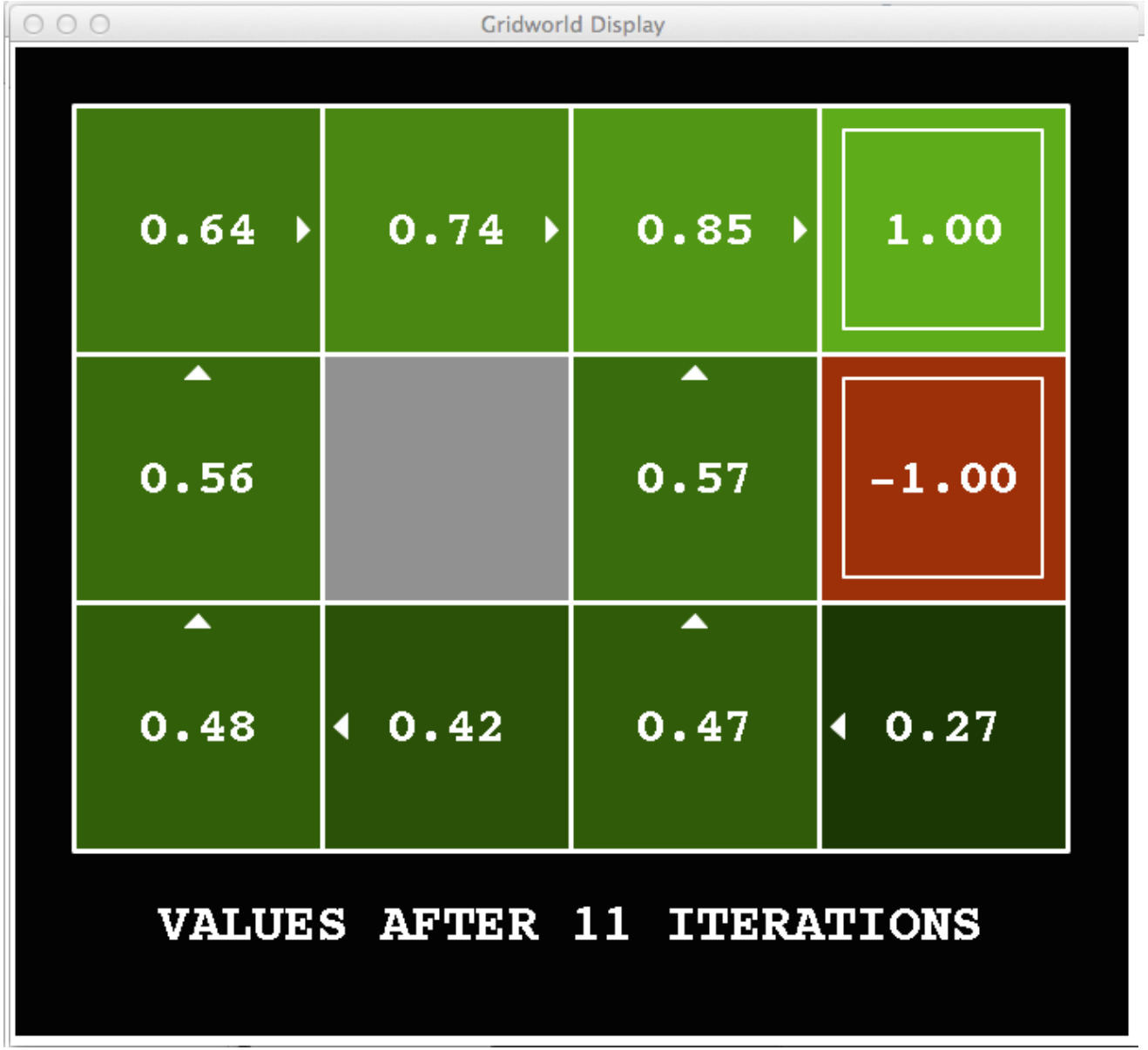
Value iteration example



Value iteration example

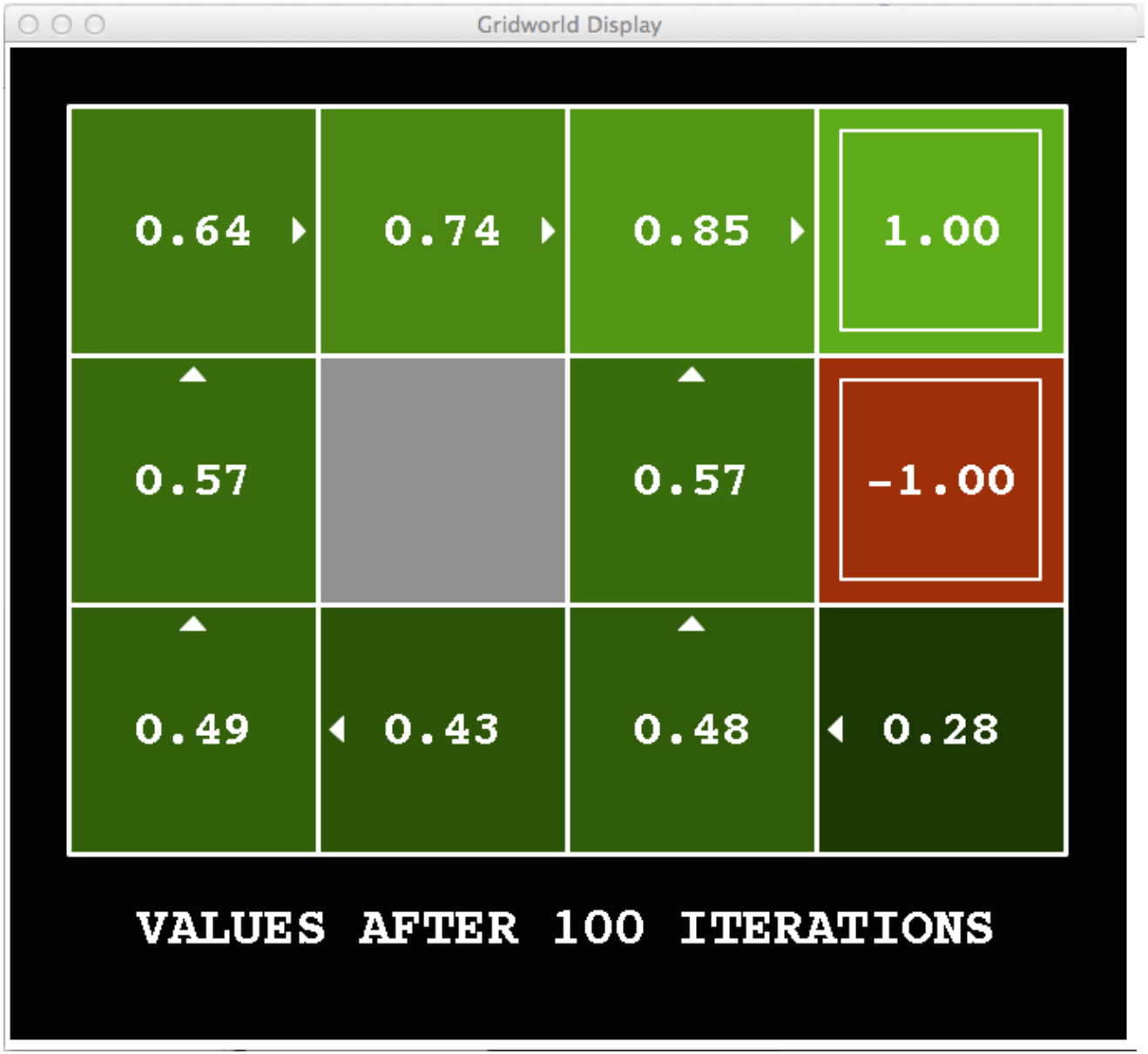


Value iteration example



Value iteration example

Value iteration example



Value iteration

Value Iteration

Input: MDP=(S,A,T,r)

Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $$V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[r(s,a) + \gamma V_i(s') \right]$$
5. if V converged, then break

Let's look at this eqn more closely...

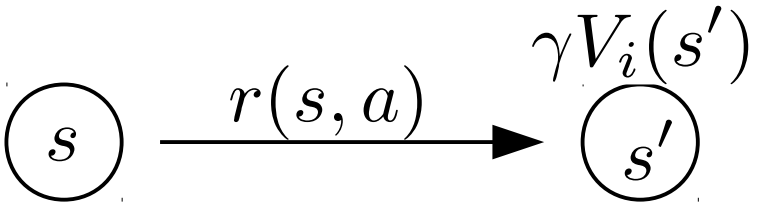
Value iteration

$$V_{i+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[r(s, a) + \gamma V_i(s') \right]$$

Value of getting to *s′* by taking *a* from *s*: $r(s, a) + \gamma V_i(s')$

reward obtained on this time step

discounted value of being at *s′*



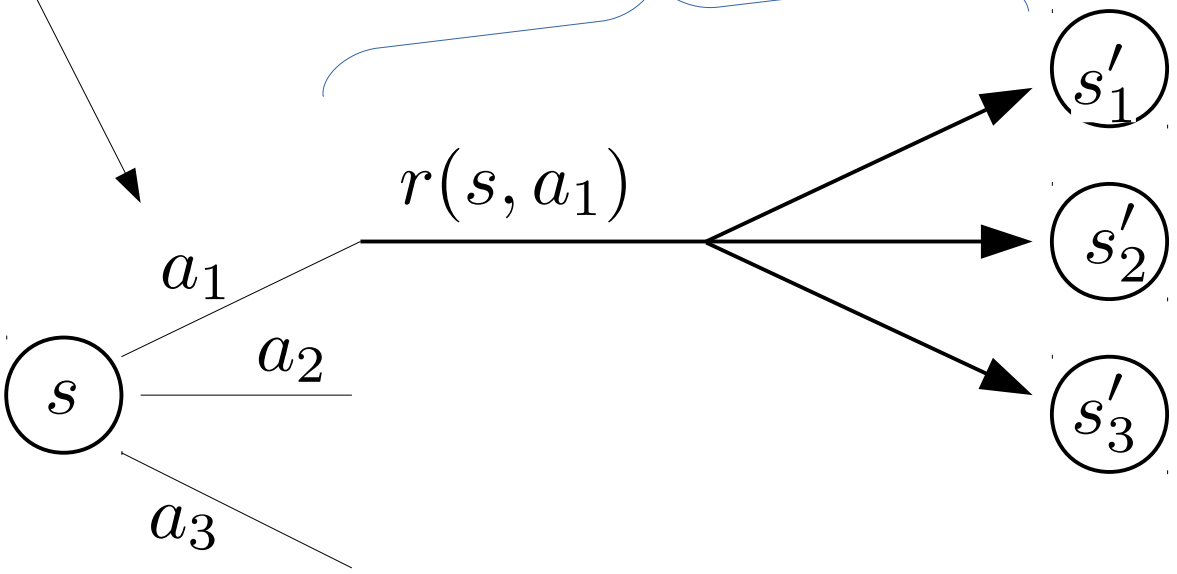
Value iteration

$$V_{i+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[r(s, a) + \gamma V_i(s') \right]$$

Value of getting to *s'* by taking *a* from *s*

Expected value of taking action *a*

Why do we maximize?



Value iteration

Value Iteration
Input: MDP=(S,A,T,r)
Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $$V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[r(s,a) + \gamma V_i(s') \right]$$
5. if V converged, then break

How do we know that this converges?

How do we know that this converges to the optimal value function?

Value iteration

At convergence, this property must hold (why?)

This is called the **Bellman Equation**

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

What does this equation tell us about optimality of *V*?

– we denote the *optimal* value function as: V^*

Gauss-Siedel Value Iteration

Value Iteration

Input: MDP=(S,A,T,r)

Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[r(s,a) + \gamma V_i(s') \right]$
5. if V converged, then break

Regular value iteration maintains two V arrays: old V and new V

Gauss-Siedel maintains only one V matrix.
– each update is immediately applied
– can lead to faster convergence

Computing a policy from the value function

Notice these little arrows

The arrows denote a policy
– how do we calculate it?

Computing a policy from the value function

In general, a policy is a distribution over actions: $\pi(s, a) : S \times A \rightarrow \mathbb{R}$

Here, we restrict consideration to deterministic policies: $\pi(s) : S \rightarrow A$

Given an optimal value function, V*, we calculate the optimal policy:

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

Optimal policy

Optimal value function

Problems with value iteration

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

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

Problem 3: The policy often converges long before the values

Policy iteration

What if you want to calculate the value function for a given sub-optimal policy?

Answer: Policy Iteration!

Value Iteration
Input: MDP=(S,A,T,r)
Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[r(s,a) + \gamma V_i(s') \right]$
5. if V converged, then break

Policy iteration

What if you want to calculate the value function for a given sub-optimal policy?

Answer: Policy Iteration!

Policy Iteration
Input: MDP=(S,A,T,r), π
Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $$V_{i+1}(s) = \sum_{s'} T(s,a,s') \left[r(s,\pi(s)) + \gamma V_i(s') \right]$$
5. if V converged, then break

Policy iteration

What if you want to calculate the value function for a given sub-optimal policy?

Answer: Policy Iteration!

Policy Iteration
Input: MDP=(S,A,T,r), π
Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $$V_{i+1}(s) = \sum_{s'} T(s,a,s') \left[r(s,\pi(s)) + \gamma V_i(s') \right]$$
5. if V converged, then break

Notice this

Policy iteration

What if you want to calculate the value function for a given sub-optimal policy?

Answer: Policy Iteration!

Policy Iteration
Input: MDP=(S,A,T,r), π
Output: value function, V

1. let $\forall s \in S, V_1(s) = V_{init}$
2. for i=1 to infinity
3. for all $s \in S$
4. $V_{i+1}(s) = \sum_{s'} T(s,a,s') \left[r(s,\pi(s)) + \gamma V_i(s') \right]$
5. if V converged, then break

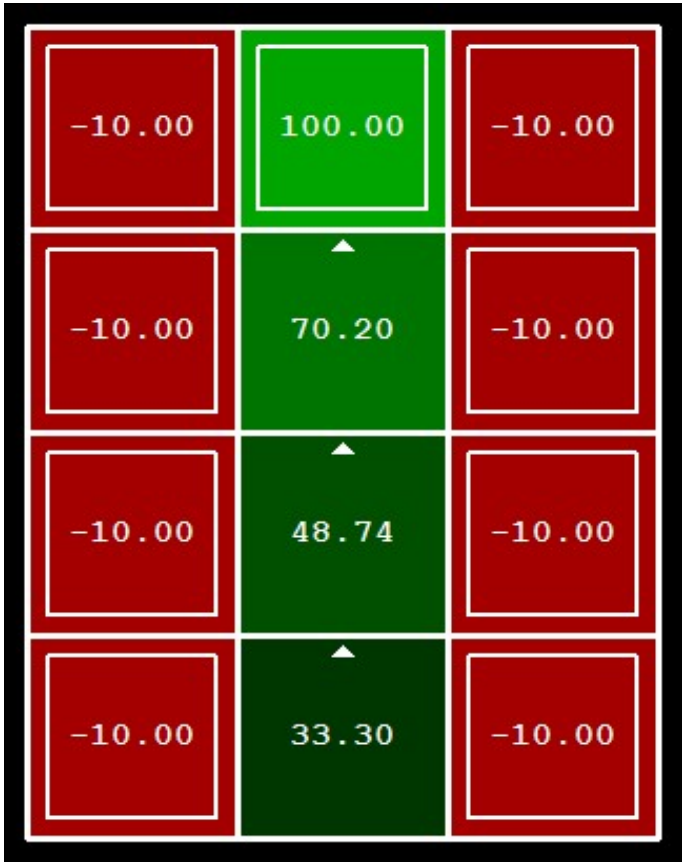
Notice this

OR: can solve for value function as the sol'n to a system of linear equations
– can't do this for value iteration because of the maxes

Policy iteration: example

Always Go Right

Always Go Forward



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

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

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*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          v ← R(s, a)
7:          for s′ ∈ S(s, a)
8:              (a′, v′) ← SelectAction(s′, d − 1)
9:              v ← v + γ T(s′ | s, a) v′
10:         if v > v*
11:             (a*, v*) ← (a, v)
12:     return (a*, v*)
```

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

Branch and bound search

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

Algorithm 4.7 Branch-and-bound search

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, U(s))
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          if Ū(s, a) < v*
7:              return (a*, v*)
8:          v ← R(s, a)
9:          for s′ ∈ S(s, a)
10:             (a′, v′) ← SelectAction(s′, d − 1)
11:             v ← v + γ T(s′ | s, a) v′
12:         if v > v*
13:             (a*, v*) ← (a, v)
14:     return (a*, v*)
```

Worse case complexity?

Monte Carlo evaluation

Algorithm 4.11 Monte Carlo policy evaluation

```
function MonteCarloPolicyEvaluation(λ, d)
    for i ← 1 to n
        s ~ b
        u_i ← Rollout(s, d, π_λ)
    return (1/n) Σ_{i=1}^{n} u_i
```

Algorithm 4.10 Rollout evaluation

```
function Rollout(s, d, π_0)
    if d = 0
        return 0
    a ~ π_0(s)
    (s', r) ~ G(s, a)
    return r + γ Rollout(s', d − 1, π_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

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, 0)
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          v ← 0
7:          for i ← 1 to n
8:              (s', r) ~ G(s, a)
9:              (a', v') ← SelectAction(s', d − 1)
10:             v ← v + (r + γv')/n
11:         if v > v*
12:             (a*, v*) ← (a, v)
13:     return (a*, v*)
```

Complexity? Guarantees?

Sparse sampling

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

Algorithm 4.8 Sparse sampling

```
1:  function SelectAction(s, d)
2:      if d = 0
3:          return (nil, 0)
4:      (a*, v*) ← (nil, −∞)
5:      for a ∈ A(s)
6:          v ← 0
7:          for i ← 1 to n
8:              (s', r) ~ G(s, a)
9:              (a', v') ← SelectAction(s', d − 1)
10:             v ← v + (r + γ v')/n
11:         if v > v*
12:             (a*, v*) ← (a, v)
13:     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, π_0)
4:      return arg max_a Q(s, a)
5:  function Simulate(s, d, π_0)
6:      if d = 0
7:          return 0
8:      if s ∉ T
9:          for a ∈ A(s)
10:             (N(s, a), Q(s, a)) ← (N_0(s, a), Q_0(s, a))
11:         T = T ∪ {s}
12:         return Rollout(s, d, π_0)
13:     a ← arg max_a Q(s, a) + c √(log N(s) / N(s, a))
14:     (s', r) ~ G(s, a)
15:     q ← r + γ Simulate(s', d − 1, π_0)
16:     N(s, a) ← N(s, a) + 1
17:     Q(s, a) ← Q(s, a) + (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