CS7IS2: Artificial Intelligence

Markov Decision Processes

So far

- Uninformed/blind search
- Informed search

- Adversarial search – still to do later

- But…all actions were deterministic → BFS, DFS

- How do we search in the presence of uncertainty? Stochastic actions

- Markov Decision Processes
  - What are they, solving them – value iteration, policy iteration, RL

Lecture Outline

- Markov property,
- Markov process, Markov reward process
- Markov decision process (MDP)

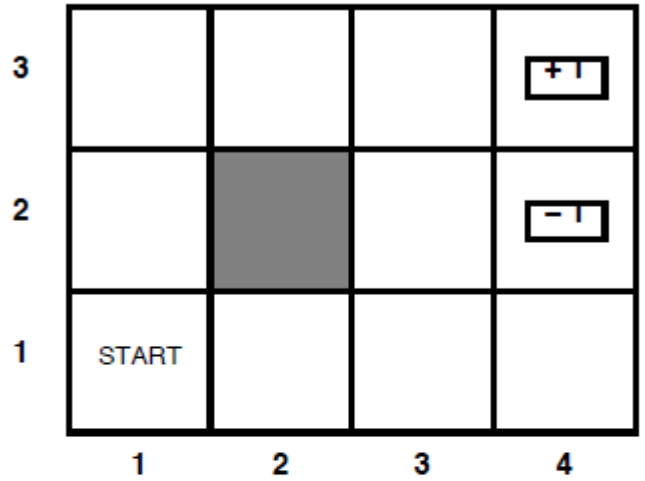
- Rewards in MDPs, discounting
- Utilities and Value functions

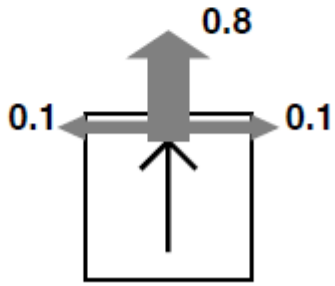
- Solving MDPs
  - Dynamic programming, Bellman Equation
  - Value Iteration, Prioritized Sweeping
- Gridworld example

- Credits: CS188 UC Berkley, UCL David Silver, course textbooks

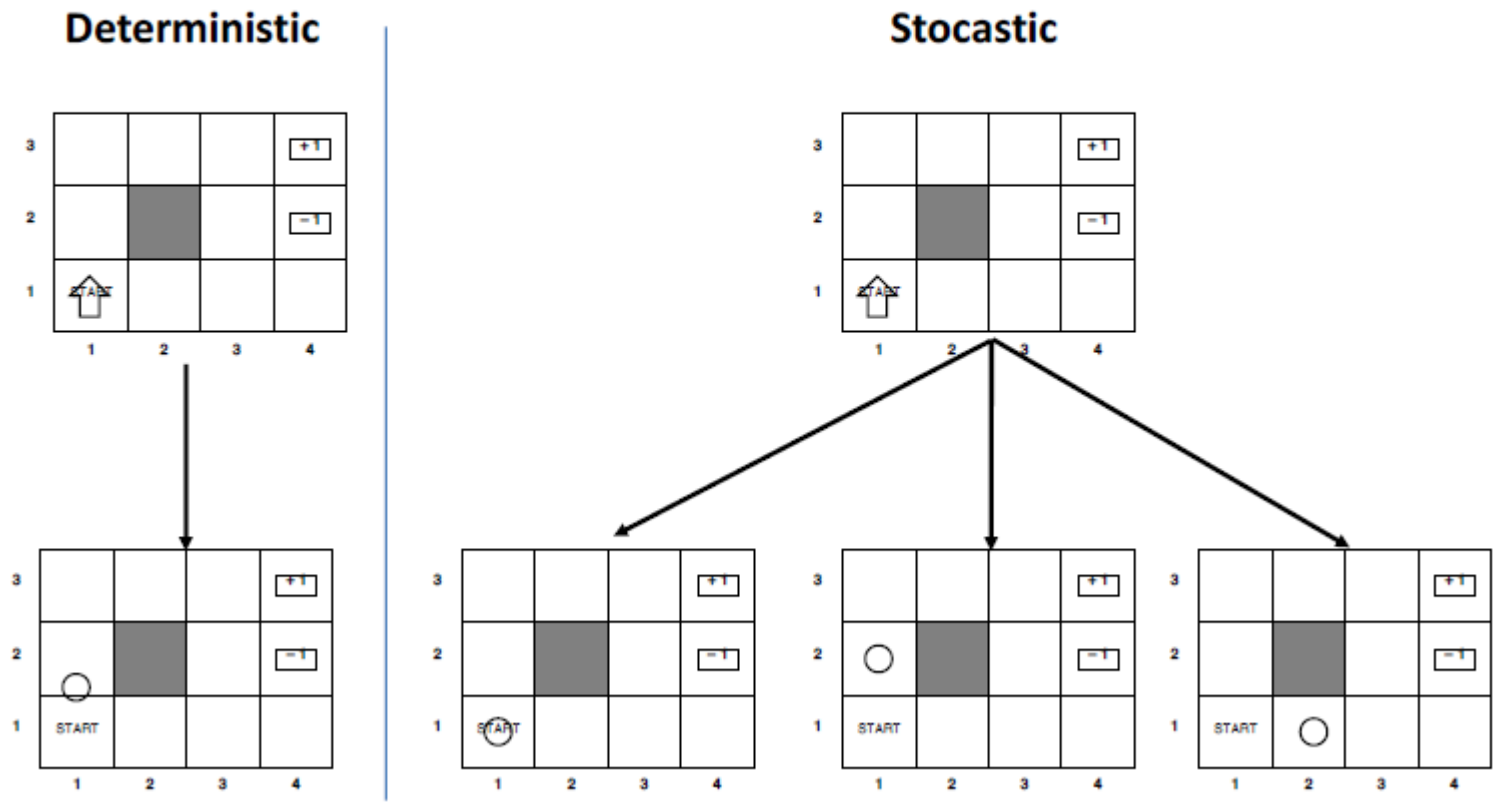
Example: Grid World

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



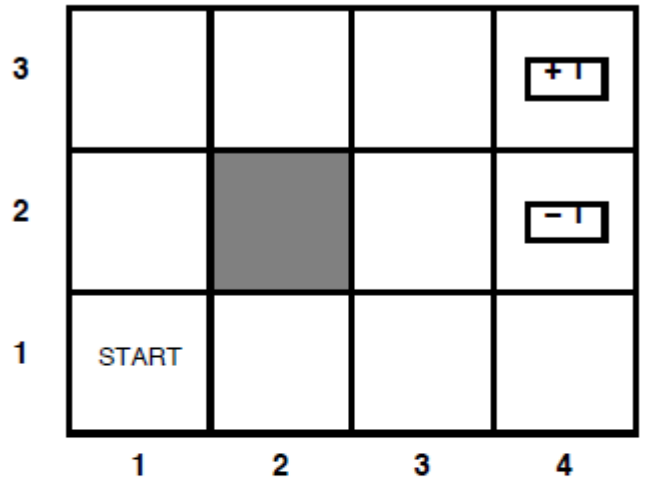


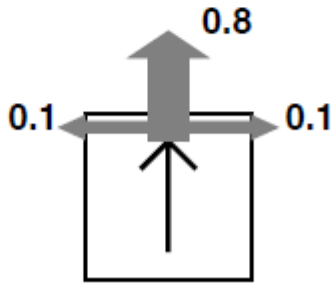
Grid world example



Markov Decision Processes

- 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
- Goal: maximize the expected reward
- MDPs are **non-deterministic search problems**
  - One way to solve them is with expectimax search (when we go back to adversarial search)





Markov Property

- "Markov" → given the present state, the future and the past are independent

Andrey Markov
(1856-1922)

- For Markov decision processes, "Markov" means action outcomes depend only **on the current state**

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

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

Only the current state and action matter.

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

Markov Process (Markov chain)

- Memoryless random process, i.e., a sequence of random states s1, s2, ... with the Markov property (i.e., in which the probability of each event depends only on the state attained in the previous event)

- Markov chain is a tuple <S, P>
  - S – finite set of states
  - P – state transition probability matrix

State Transition Matrix

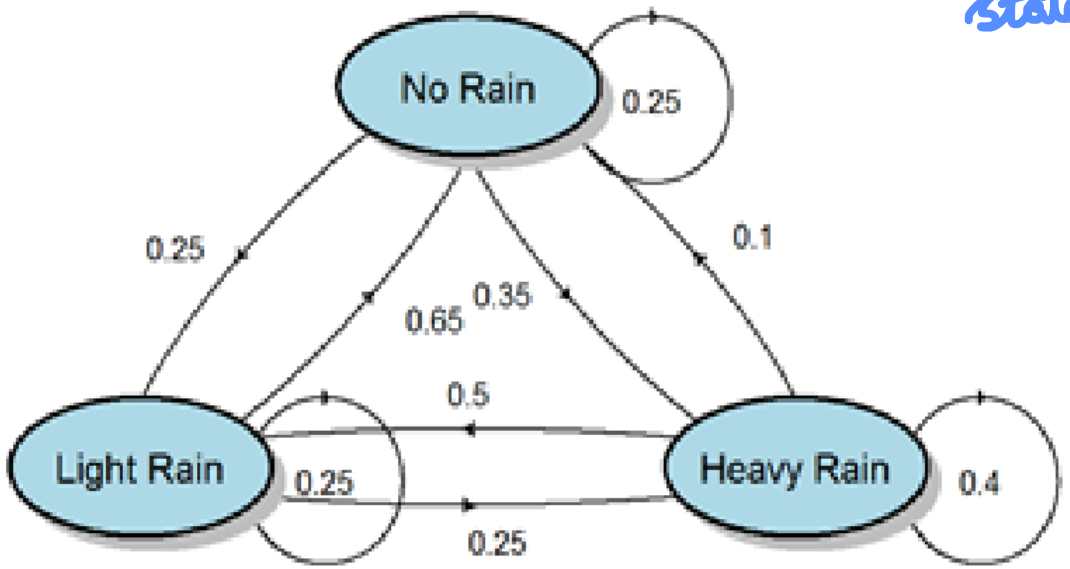
- The **state transition** probability **matrix** of a **Markov chain** gives the probabilities of transitioning from one **state** to another in a single time unit, form all states S to all successor states s'
- Each row of the matrix sums to 1

$$\mathcal{P} = \begin{matrix} \\ from \end{matrix} \begin{matrix} \overset{to}{} \\ \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \end{matrix}$$

$$\begin{matrix} & \text{Hot} & \text{Cold} \\ \text{Hot} \\ \text{Cold} \end{matrix} \begin{pmatrix} 0.2 & 0.8 \\ 0.6 & 0.4 \end{pmatrix} \begin{matrix} = 1.0 \\ = 1.0 \end{matrix}$$

Example Markov Chain

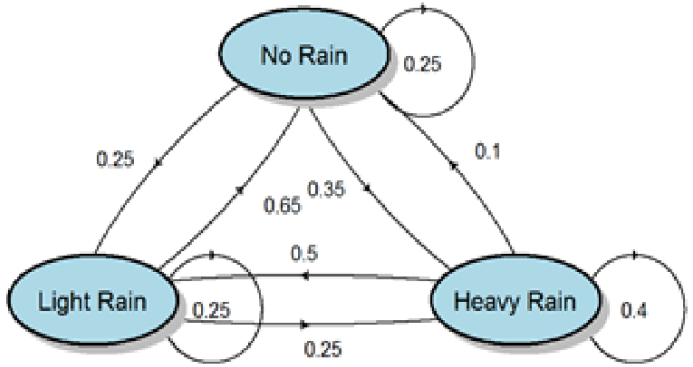
Probabilities represent movement between states



Markov Chain Episode

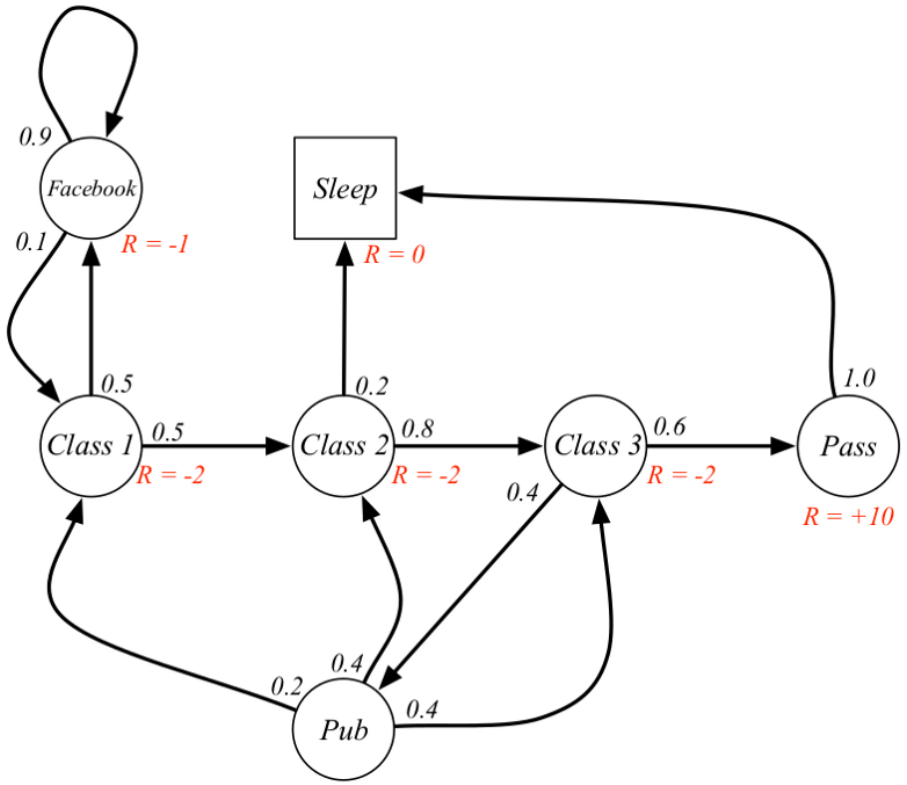
- $S_1, S_2,...S_t$

- Episode 1: Light rain, Heavy rain, Light rain
- Episode 2: Light rain, Heavy rain, Heavy rain
- …



Markov Reward Process

- Markov chain with values
- A Markov Reward Process is a tuple <S, P, R, γ>
  - S – finite set of states
  - P – state transition probability matrix
  - R - a reward function, R (s, s’)
  - γ - a discount factor [0, 1]



Markov Decision Process

- Markov reward process with decisions
- A Markov Reward Process is a tuple <S, A, P, R, γ>
  - S – finite set of states
  - A – finite set of actions
  - P – state transition probability matrix
  - R - a reward function, R (s, a, s’)
  - γ - a discount factor [0, 1]

An example MDP – recycling robot

- Actions:
  - (1) actively search for a can,
  - (2) remain stationary and wait for someone to bring it a can
  - (3) go back to home base to recharge its battery
- A(high) = {search, wait}
- A(low) = {search, wait, recharge}.

Recycling robot MDP

s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	r_{wait}
low	wait	high	0	r_{wait}
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	0.

S = state of battery

a = actions it can take

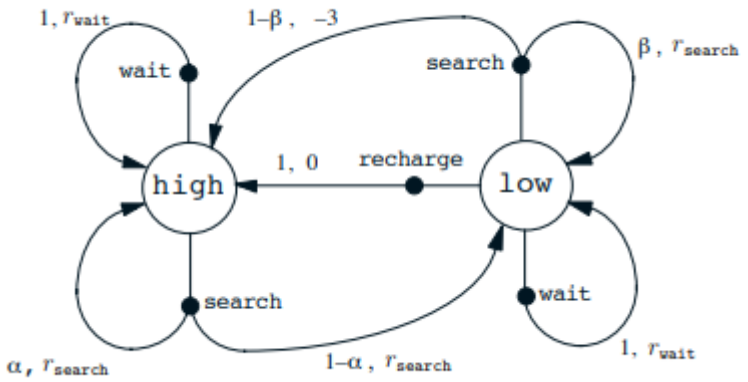
S' = next state

P(s'|s, a) = probability of the next state given the current state and action

r = reward

Recycling robot MDP

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1-\alpha$	$r_{\texttt{search}}$
low	search	high	$1-\beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	$r_{\texttt{wait}}$
low	wait	high	0	$r_{\texttt{wait}}$
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	0.



Solving MDPs: Policies

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

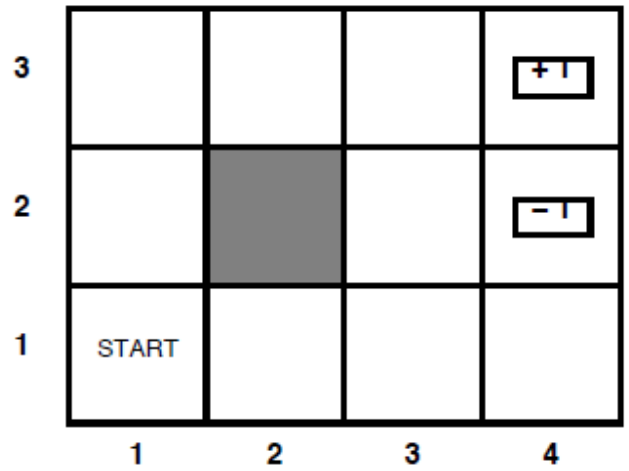
We used DFS, BFS which followed the rules of a plan.

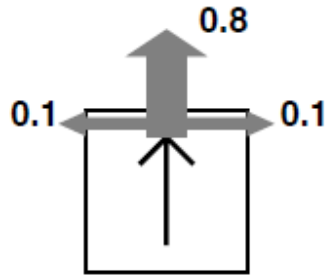
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

MDP follows a policy that gives an action for each given state.

Optimal Policies

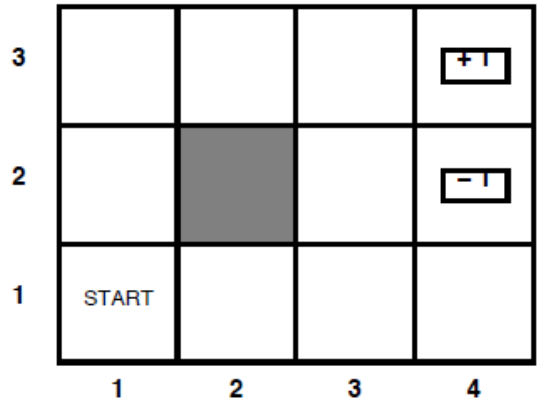
- What is the optimal policy?
- Exit states +1, -1
- Pay attention to transition probabilities!
  - 80% chance of keeping straight
  - 10% chance of turning right
  - 10% chance of turning back
  - 0% chance of going backwards

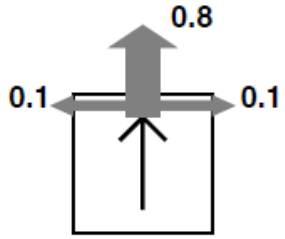




Optimal Policies

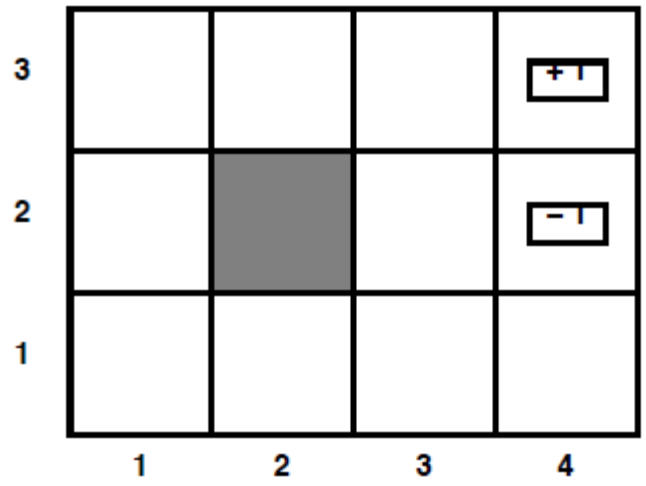
- What about if we introduce additional “living reward” at each time step?
  - R (s) = - 0.01
  - R (s) = - 0.03
  - R (s) = - 0.4
  - R (s) = -2.0



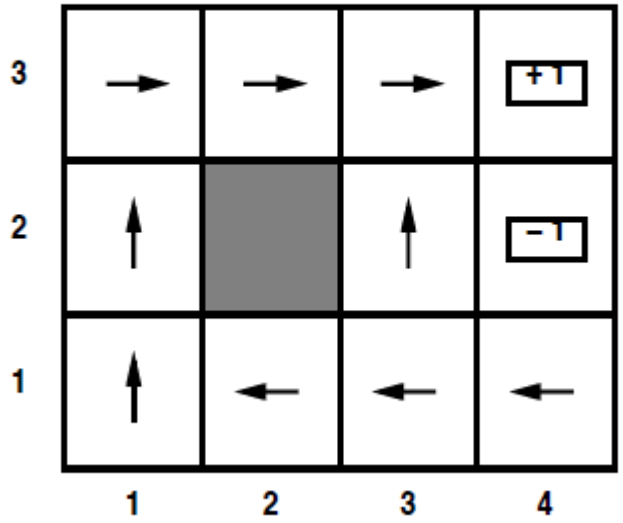


Optimal policies

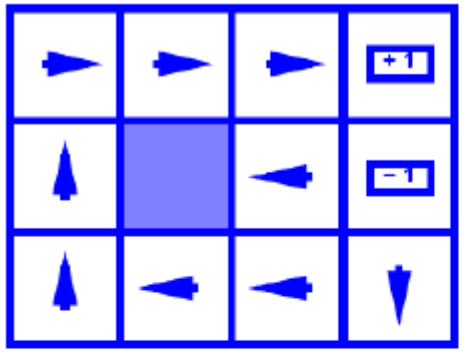
$R(s) = -0.04$



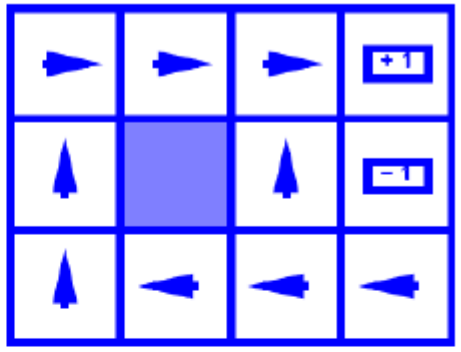
$R(s) = -0.04$



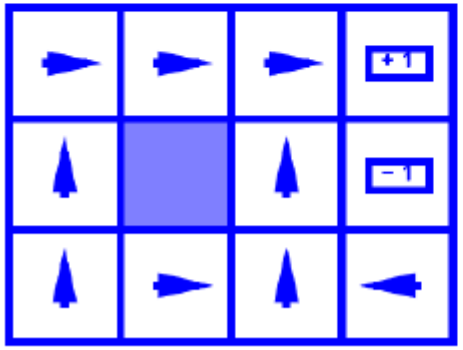
Optimal policies



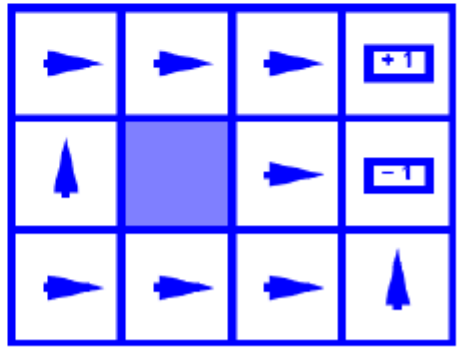
R(s) = -0.01



R(s) = -0.03



R(s) = -0.4

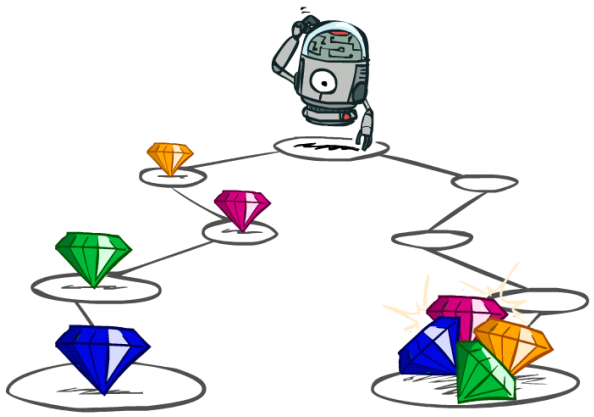


R(s) = -2.0

Goal: maximize the reward

› but since it is a sequence of rewards, what preferences should an agent have over sequence of rewards?

More or less?	[1, 2, 2]	or	[2, 3, 4]
Now or later?	[0, 0, 1]	or	[1, 0, 0]



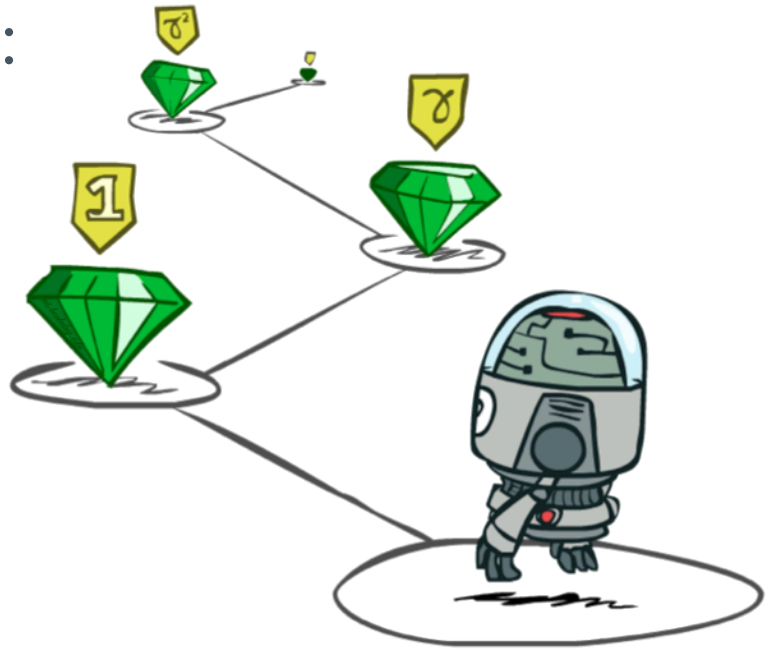
Stationary Preferences

› 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:

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

Reward discounting

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

Reward discounting

How to discount?

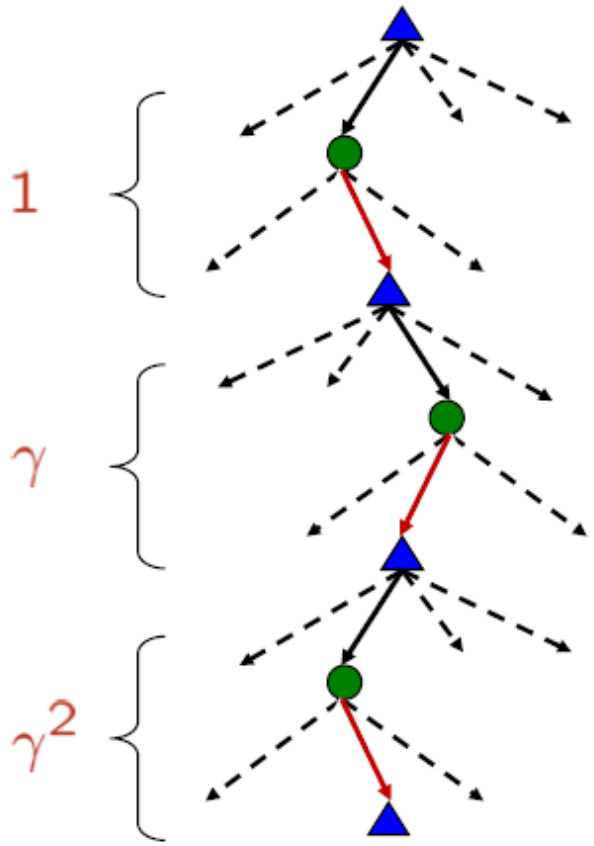
- Each time we descend a level, we multiply in the discount once

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])



Reward Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Think of it as a γ chance of ending the process at every step
  - Helps algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([3,2,1]) = 1*3+0.5*2+0.25*1
  - U([1,2,3]) < U([3,2,1])

Discounting

Given:

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

Transitions: deterministic

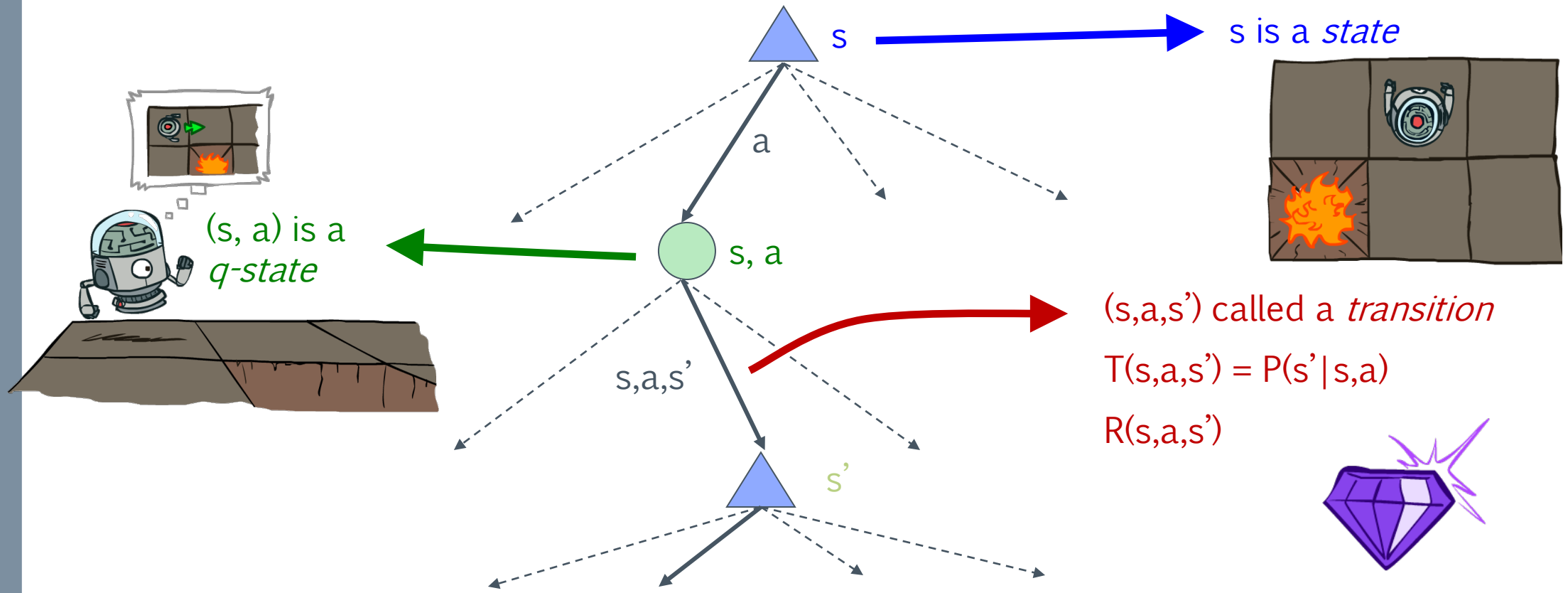
Quiz 1: For $\gamma = 1$, what is the optimal policy?

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

Is there a γ for which the actions E and W are equally good in state d?

MDP Search Trees

- Each MDP state projects a search tree

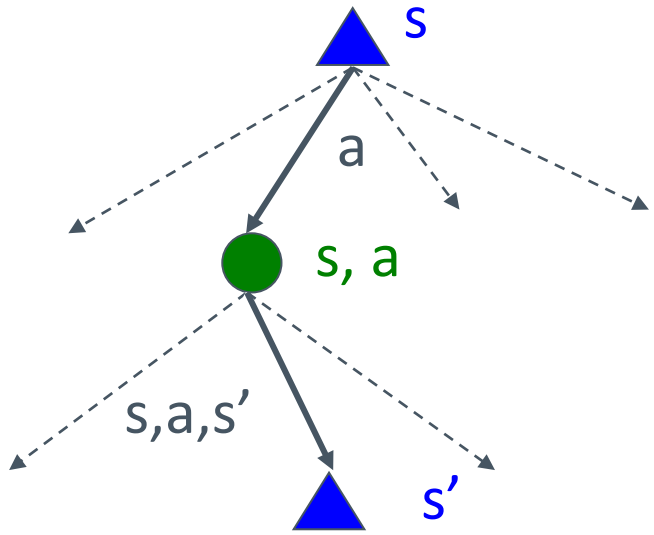


Infinite Utilities

- What if the process 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: $0 < \gamma < 1$
    - Smaller γ means smaller "horizon" – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached

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)



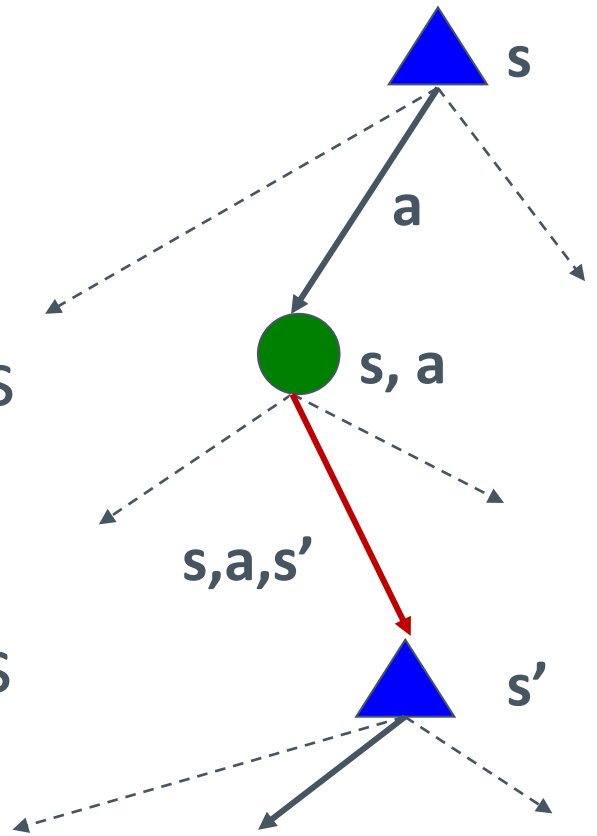
- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

Solving MDPs: Definitions

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

  $V_\pi(s)$ = expected utility starting in s and then following policy π

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



s is a *state*

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

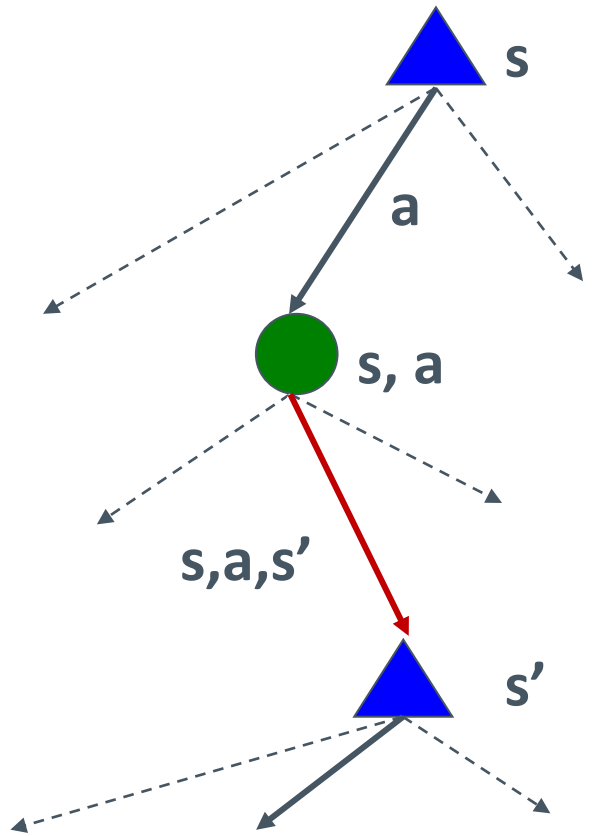
(s,a,s') is a *transition*

Solving MDPs: Definitions

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

  $Q_\pi(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) following policy π

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



s is a *state*

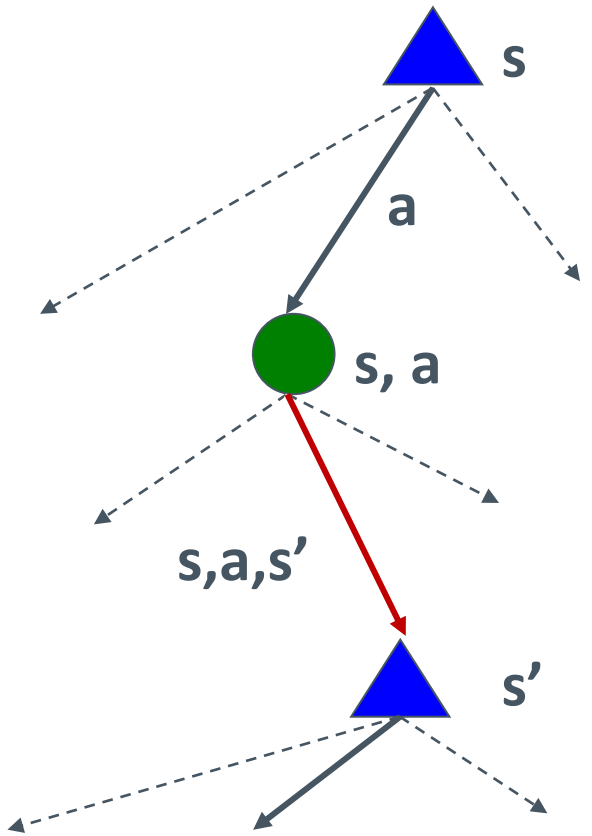
(s, a) is a *q-state*

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

Solving MDPs: Definitions

- 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*

Solving MDPs

- The optimal state-value function v*(s) is the maximum value function over all policies

$$v_*(s) = \max_\pi v_\pi(s)$$

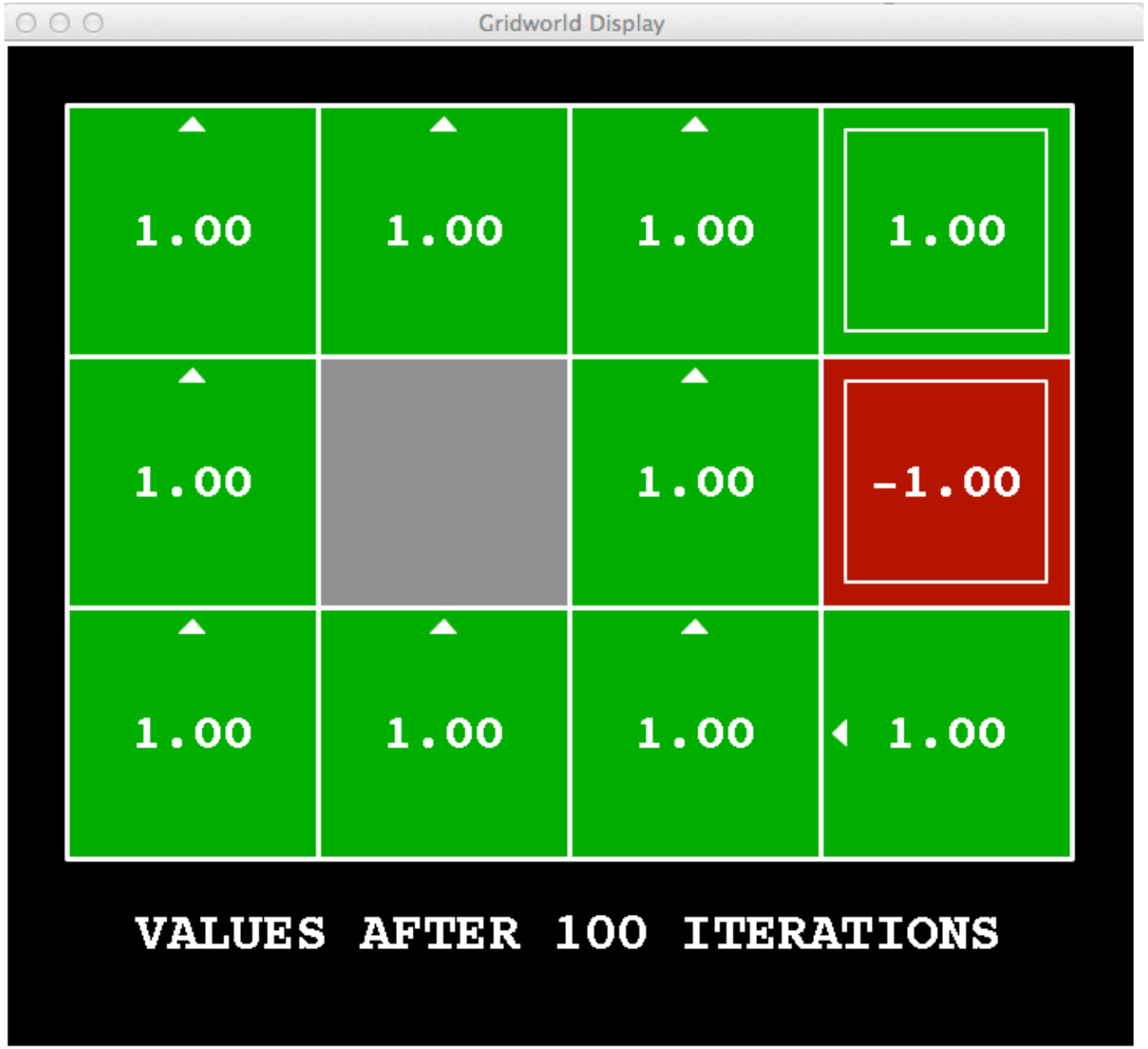
- The optimal action-value function q*(s, a) is the maximum action-value function over all policies

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

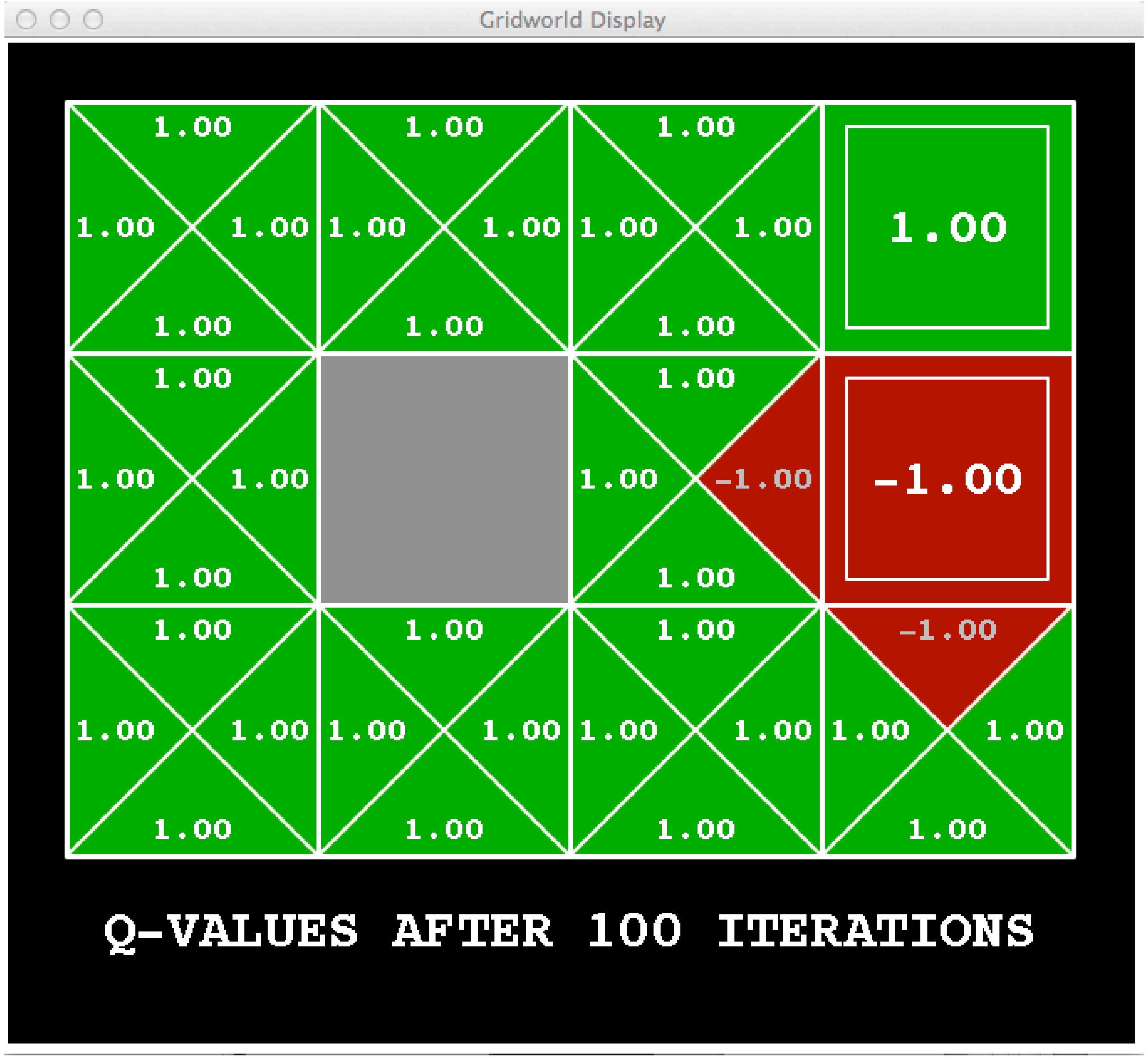
- The optimal value function species the best possible performance in the MDP.
- An MDP is “solved" when we know the optimal value function.

Solving MDPs: Parameters

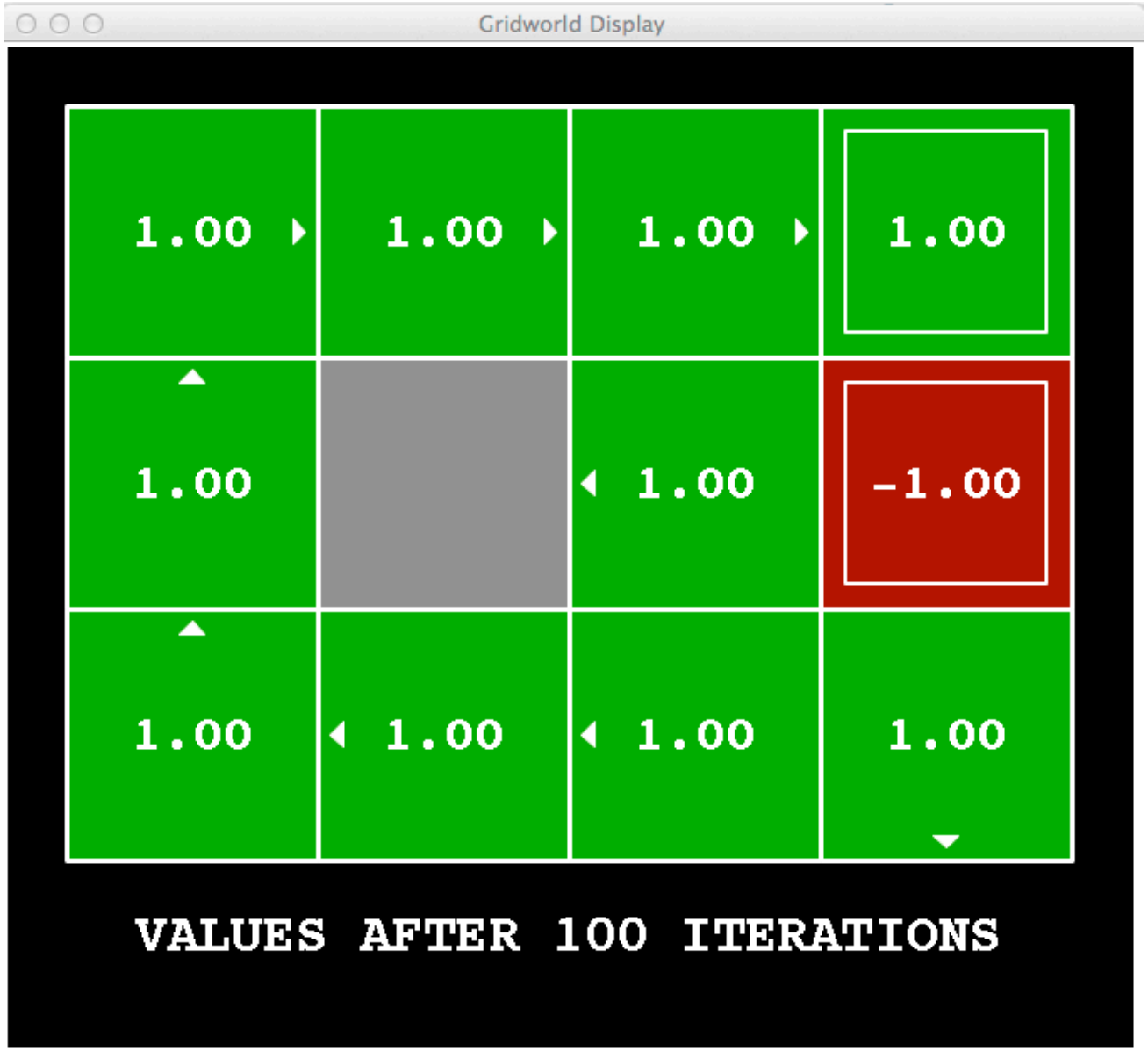
- Noise
- Living reward
- Discount



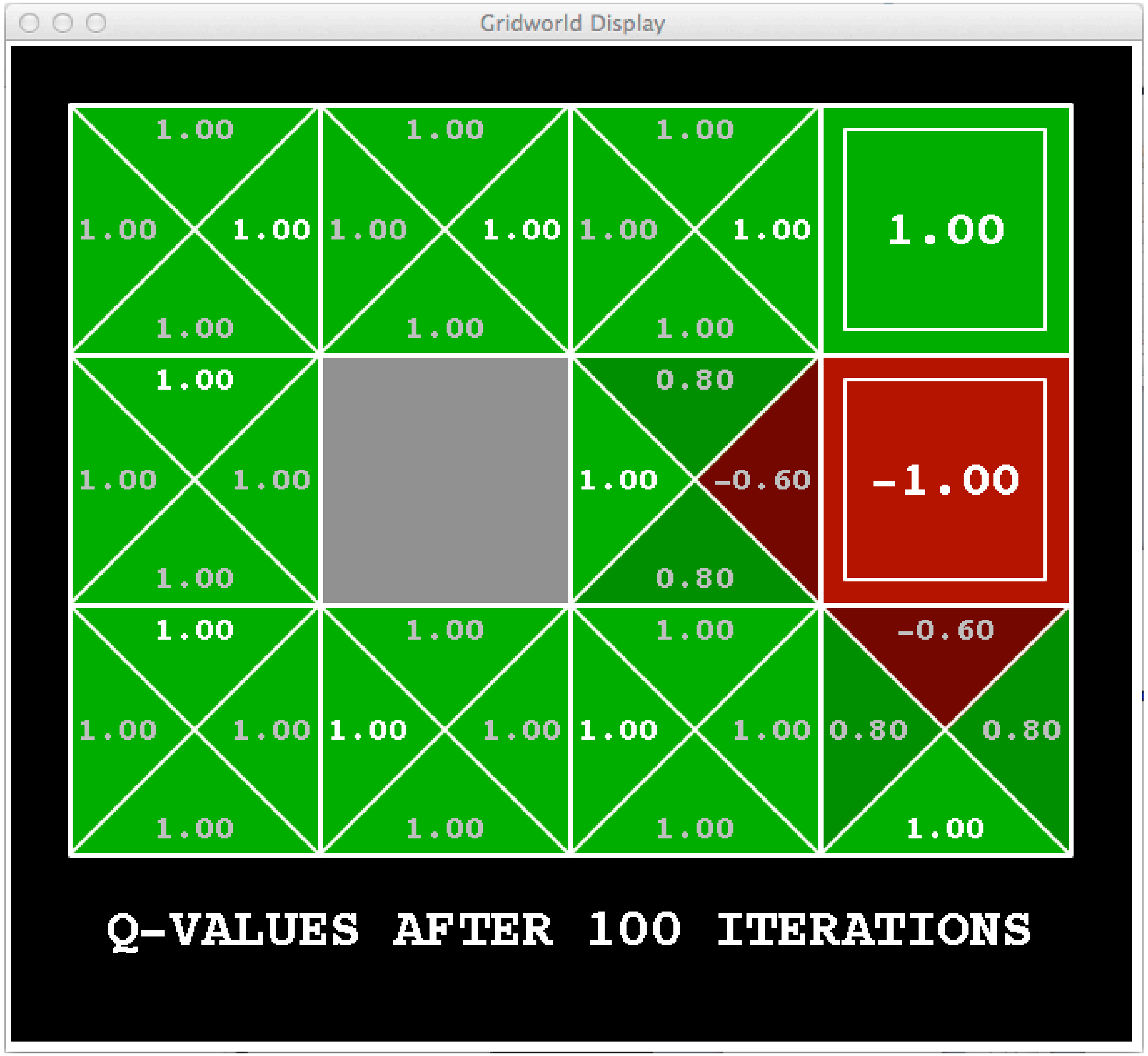
Noise = 0
Discount = 1
Living reward = 0



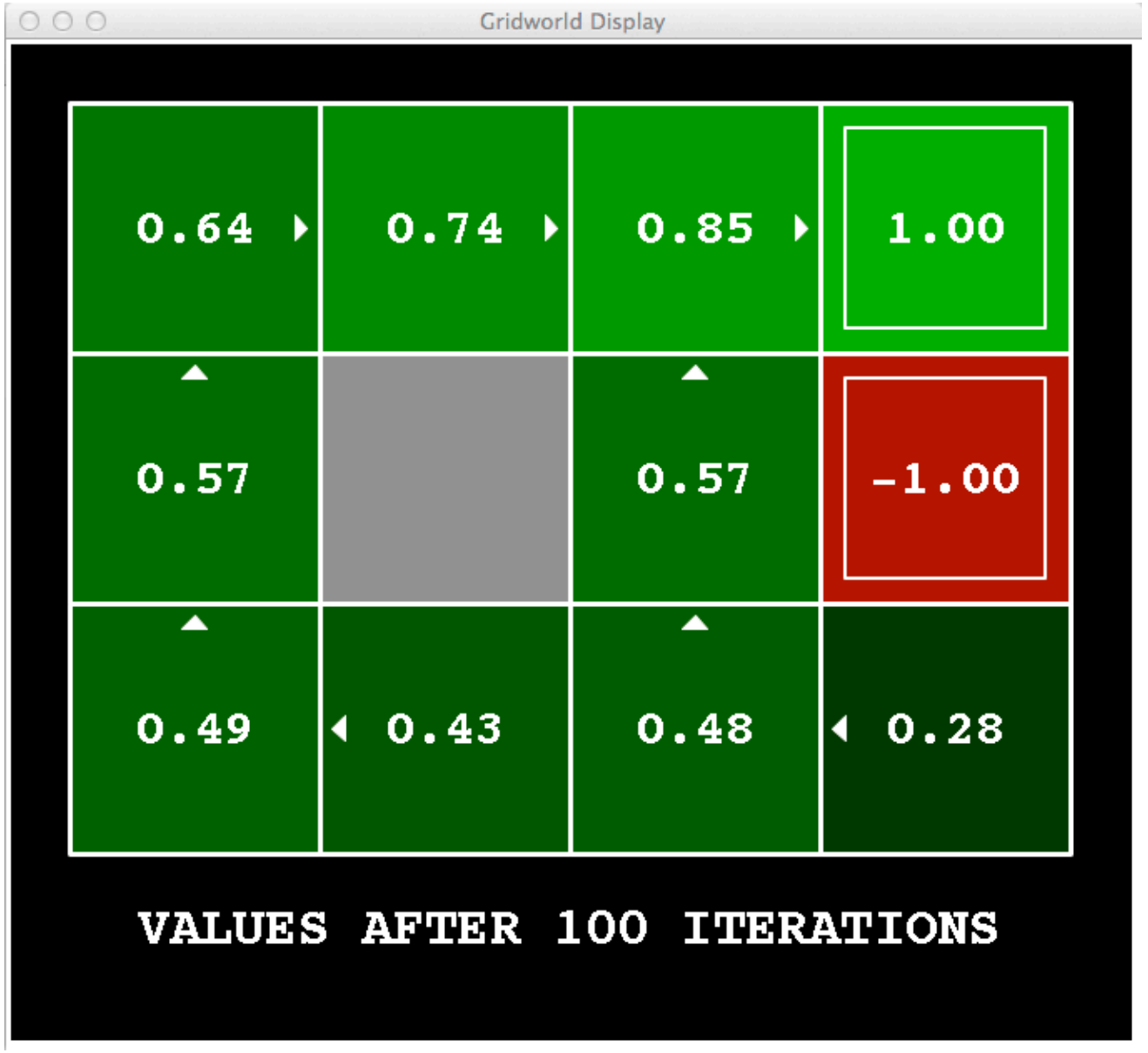
Noise = 0
Discount = 1
Living reward = 0



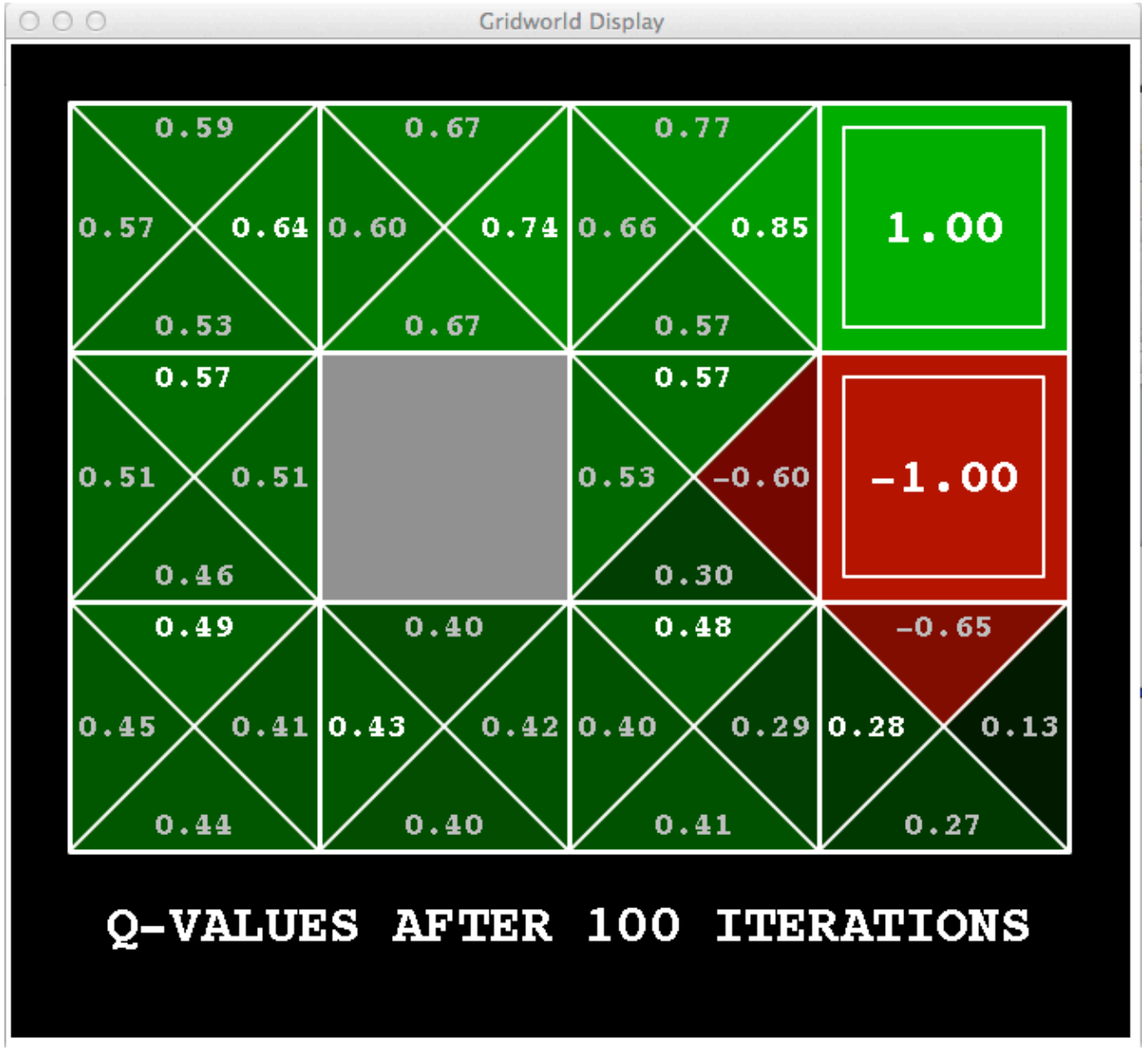
Noise = 0.2
Discount = 1
Living reward = 0



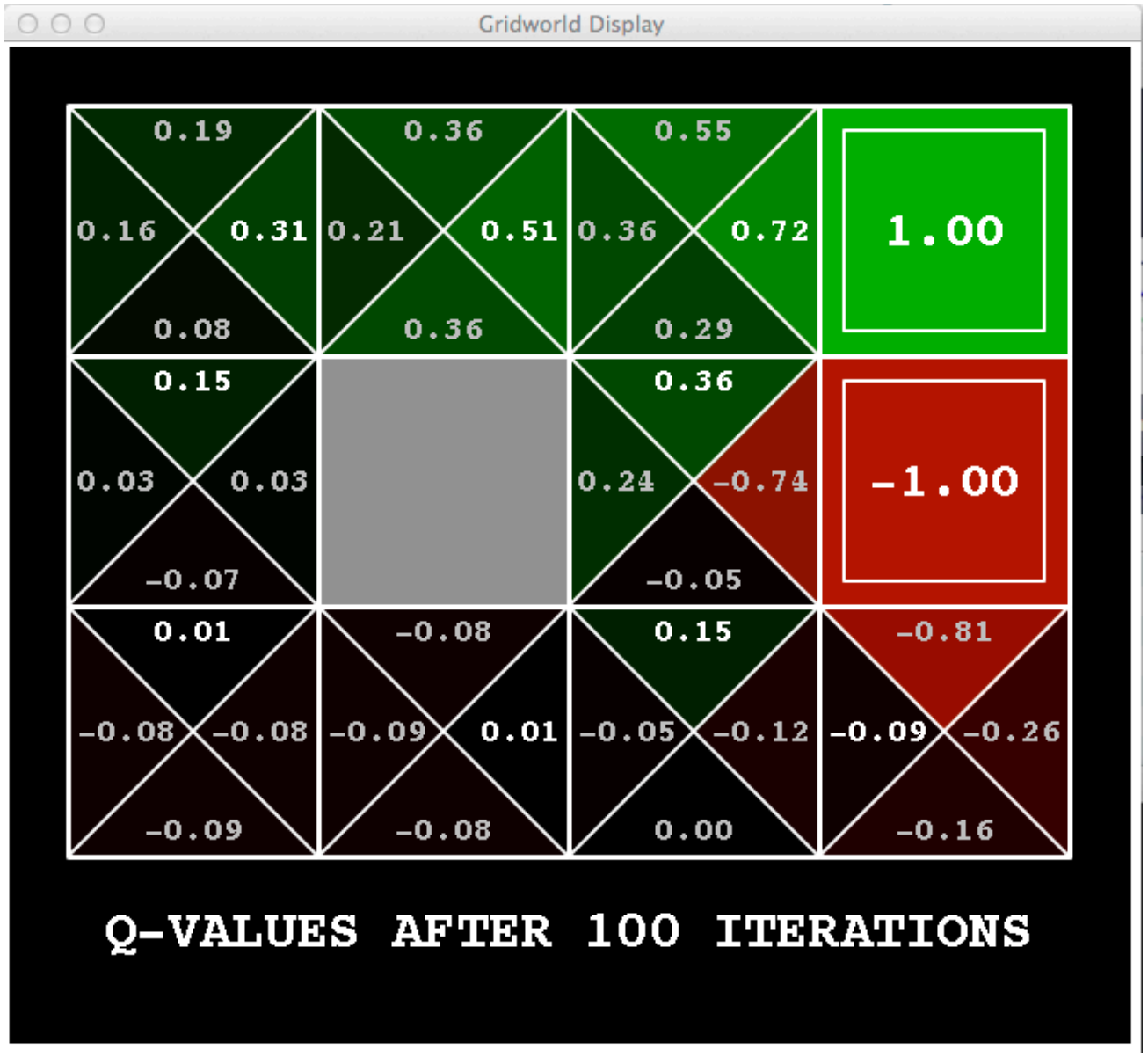
Noise = 0.2
Discount = 1
Living reward = 0



Noise = 0.2
Discount = 0.9
Living reward = 0



Noise = 0.2
Discount = 0.9
Living reward = 0



Noise = 0.2
Discount = 0.9
Living reward = -0.1

Optimal policy

- For any Markov Decision Process

- There exists an optimal policy that is better than or equal to all other policies
- All optimal policies achieve the optimal value function
- All optimal policies achieve the optimal action-value function,

For MDP, there is always a solution because the goal is to maximize the reward.

Finding optimal policy

› An optimal policy can be found by maximizing over q*(s, a)

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s, a) \\ 0 & \textit{otherwise} \end{cases}$$

› There is always a deterministic optimal policy for any MDP

› If we know q*(s, a) we immediately have the optimal policy

Bellman Equation

- Fundamental operation – compute the value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
- Recursive definition of value:

V = value
T = transition
R = reward

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

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

Bellman Equation

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

Transition Reward discount Value

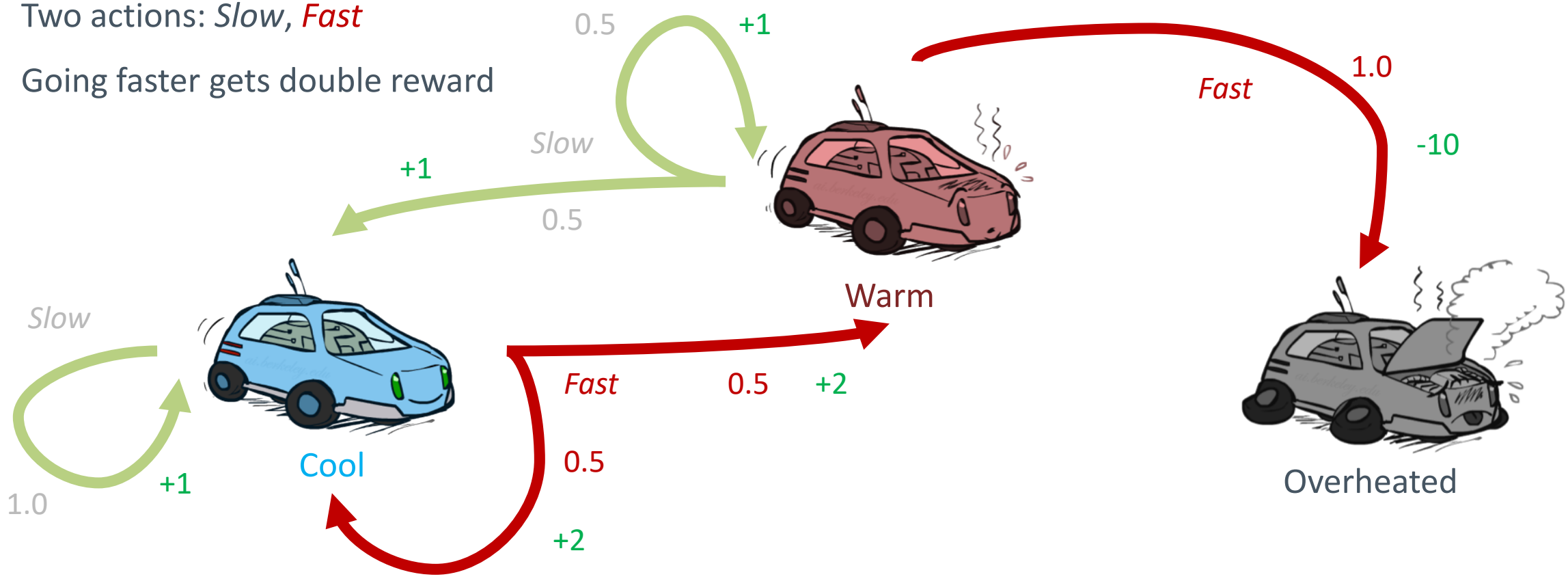
The Bellman equations

How to be optimal:

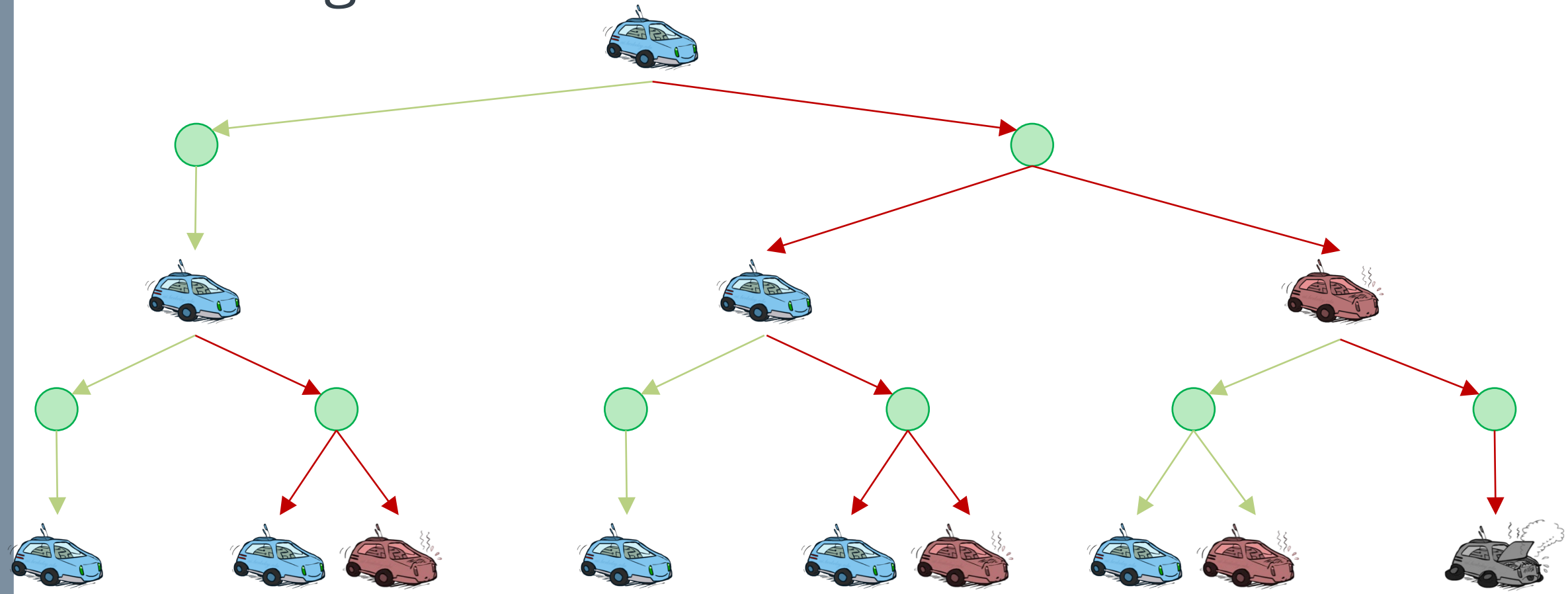
1. Take the correct first action
2. Keep being optimal

Example: Racing

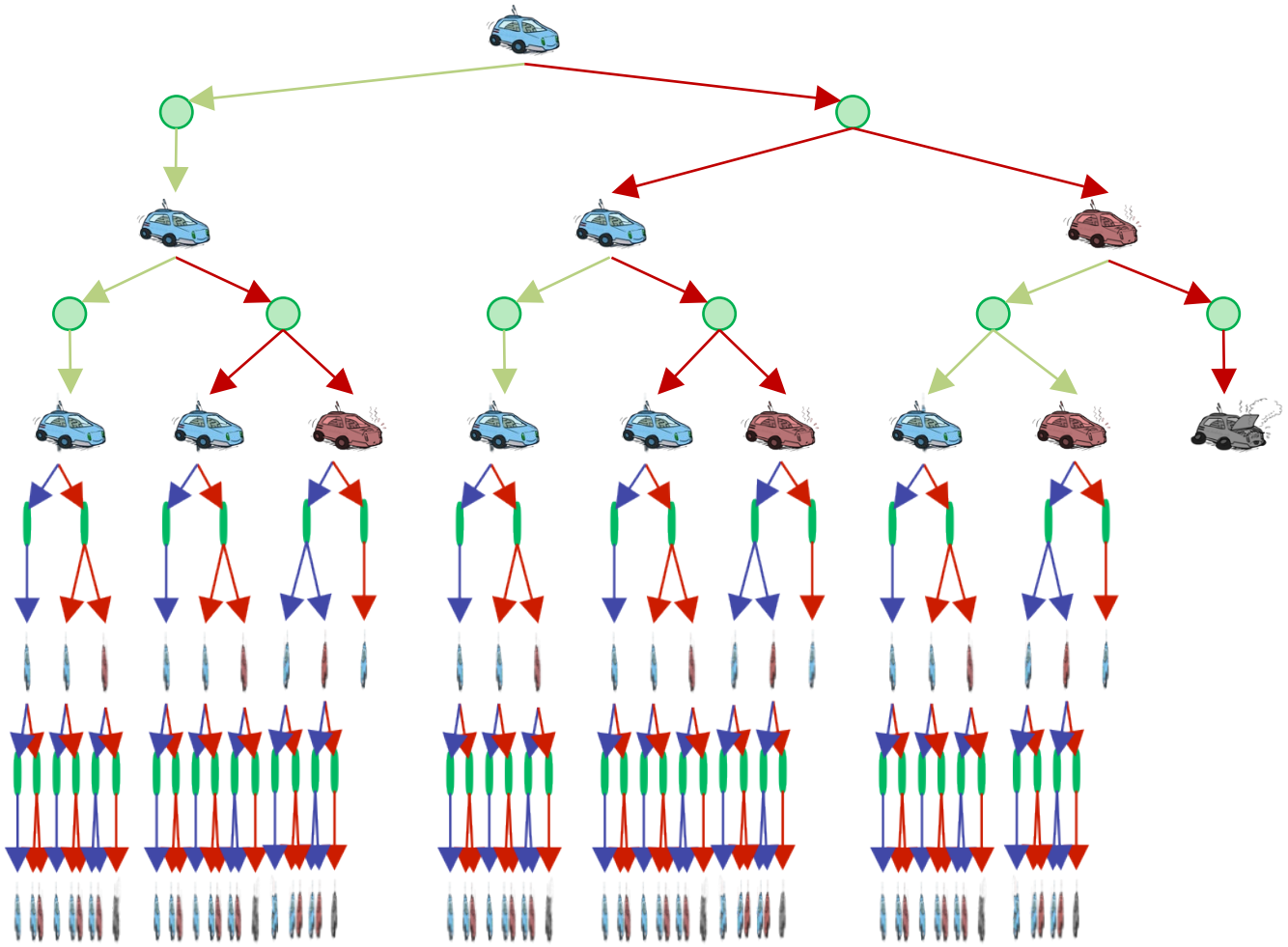
- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward



Racing Search Tree

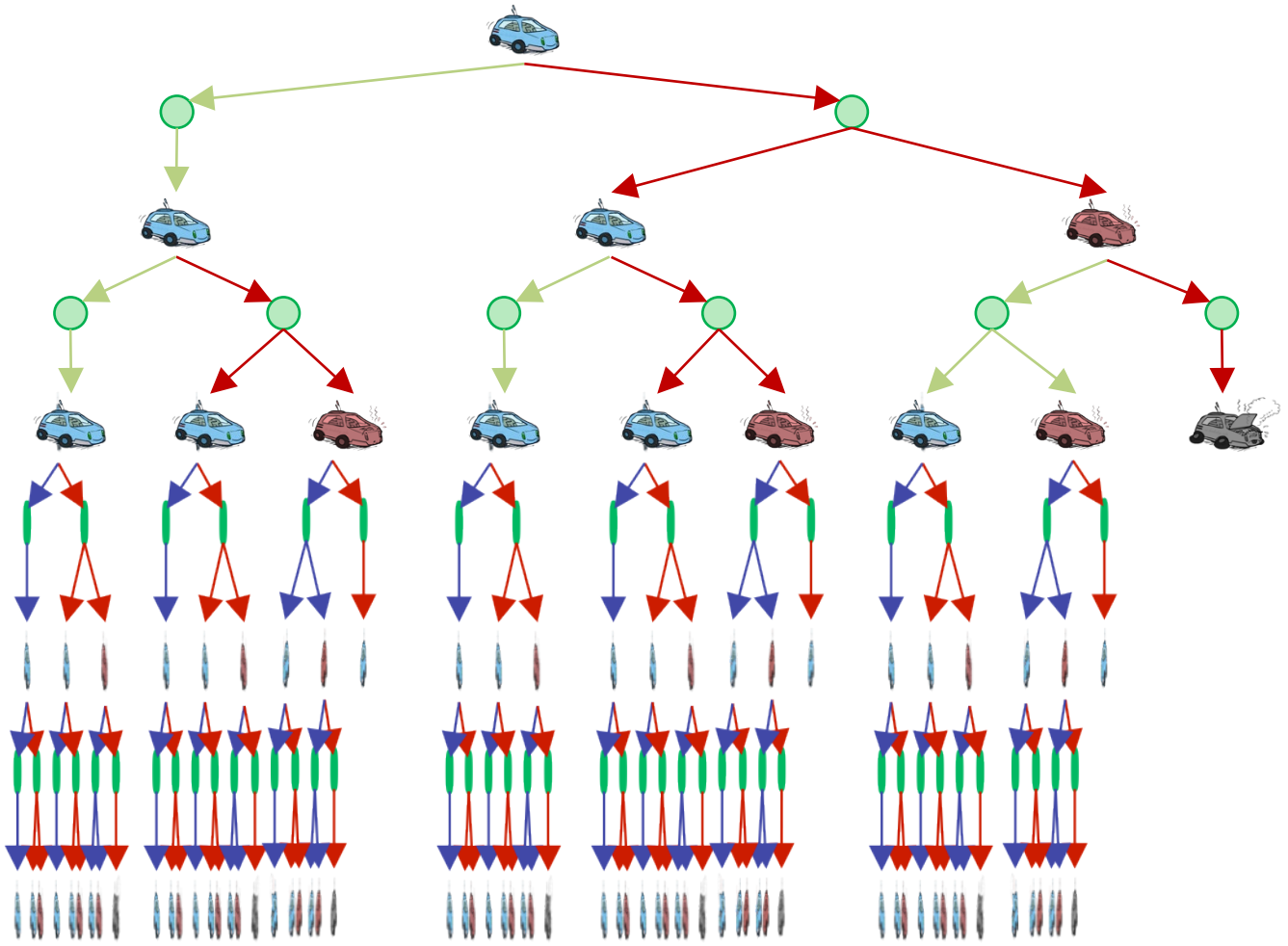


Racing Search Tree



Racing Search Tree

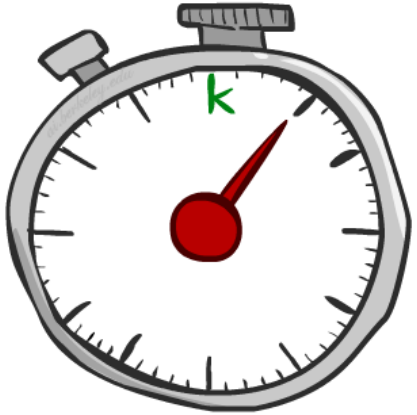
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter if $\gamma < 1$

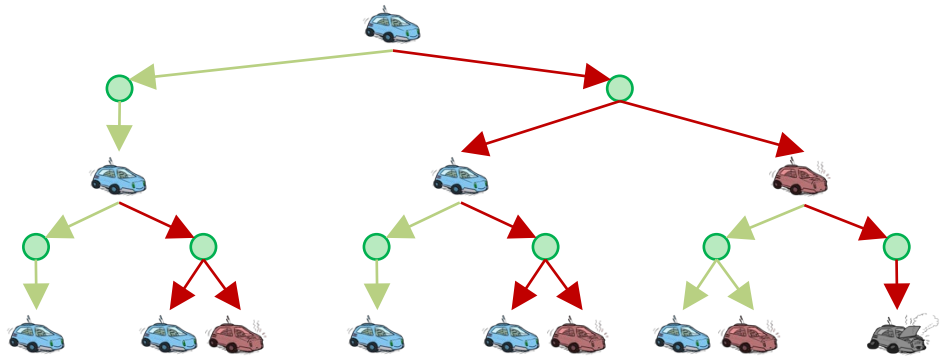


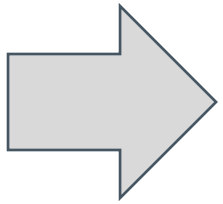
Value Iteration

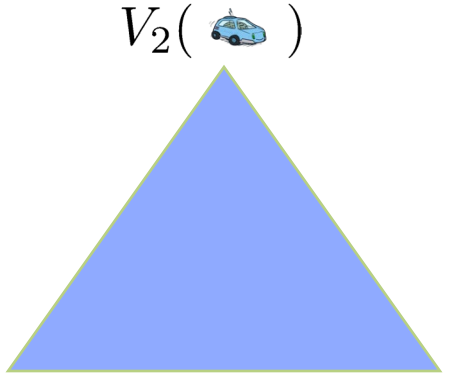
Time-Limited Values

- 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

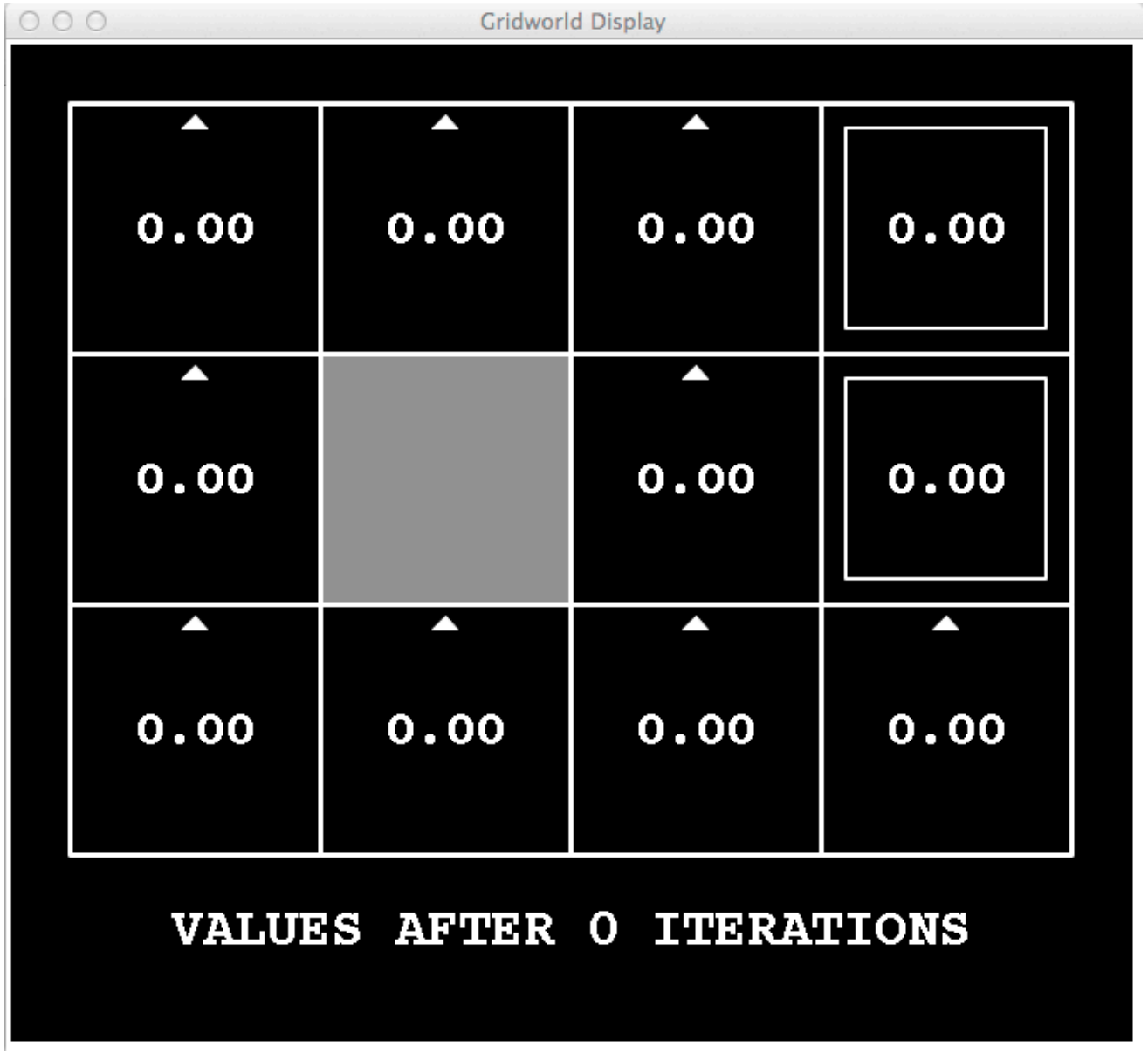






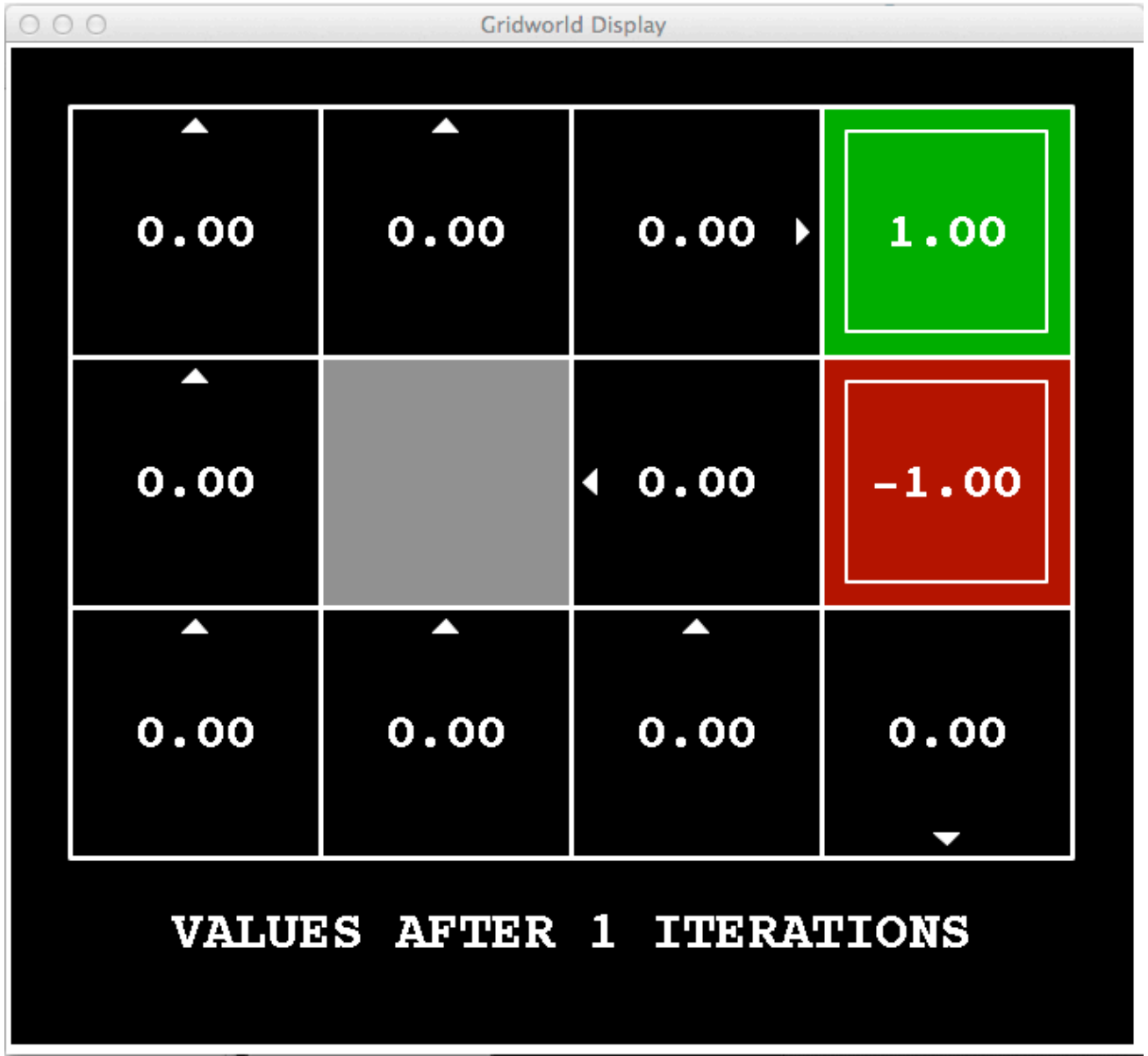


k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1

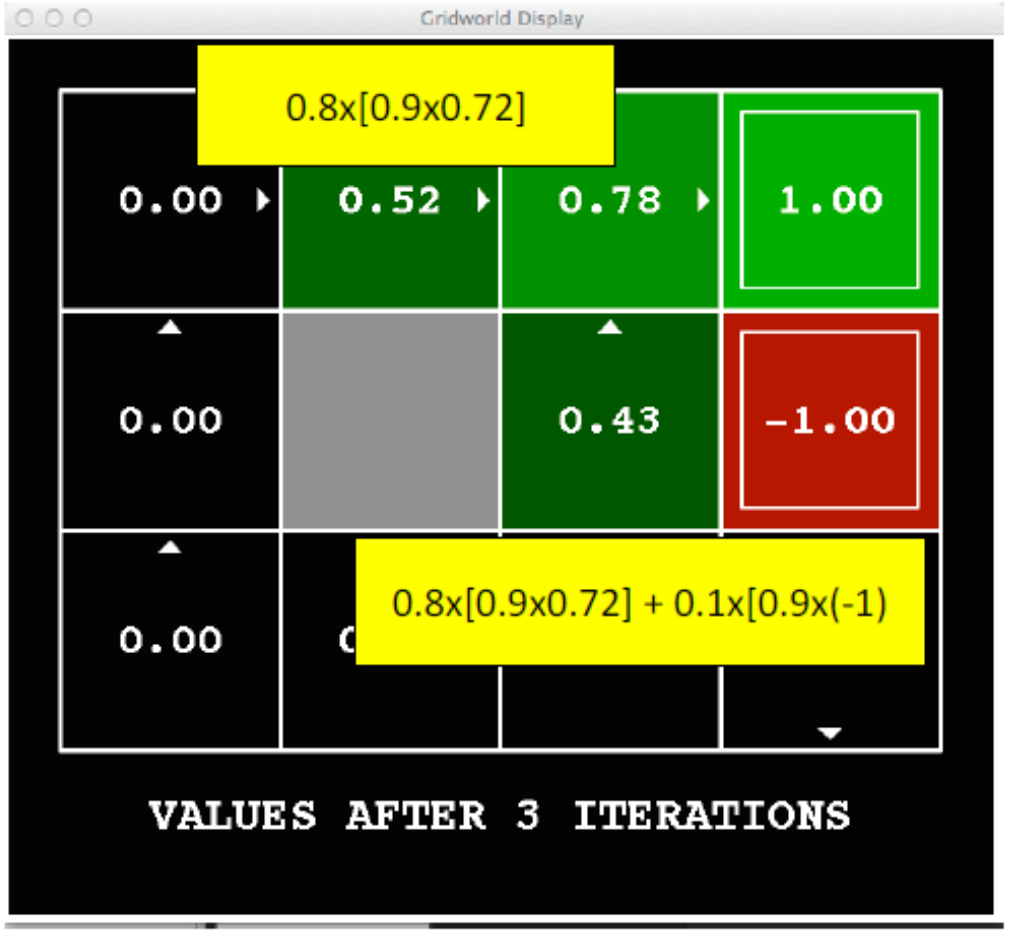


Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

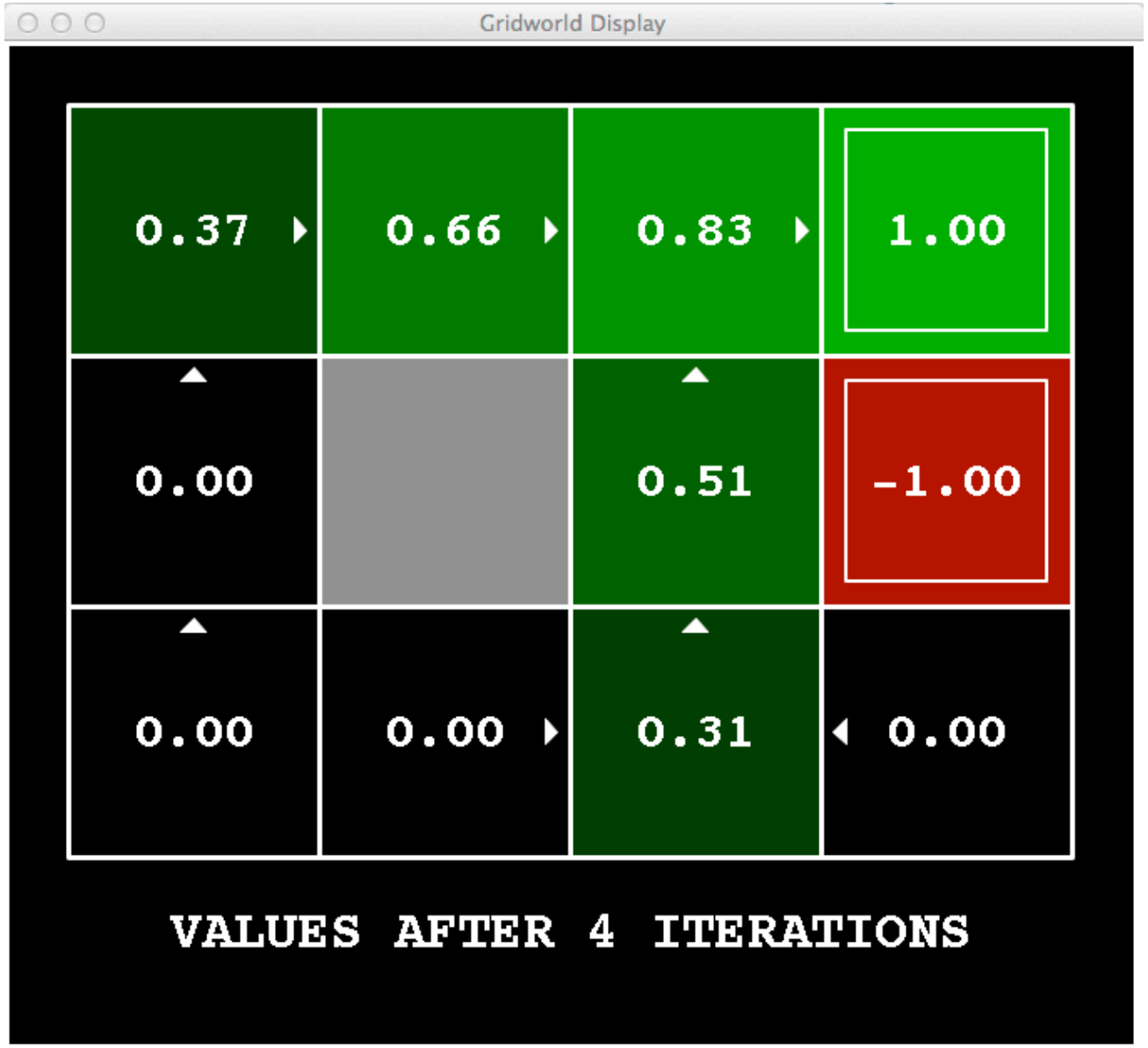
Noise = 0.2
Discount = 0.9
Living reward = 0

k=3



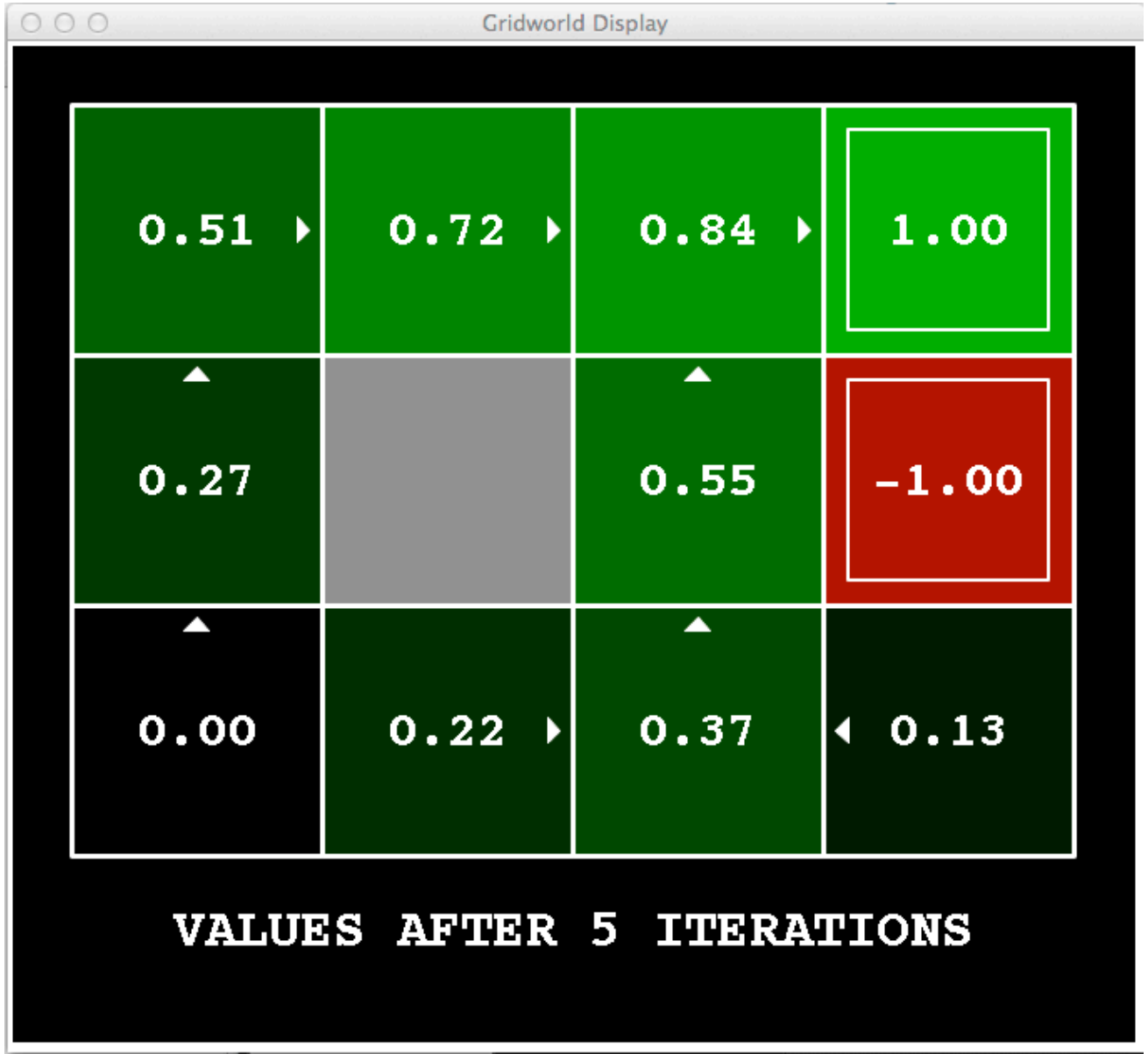
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



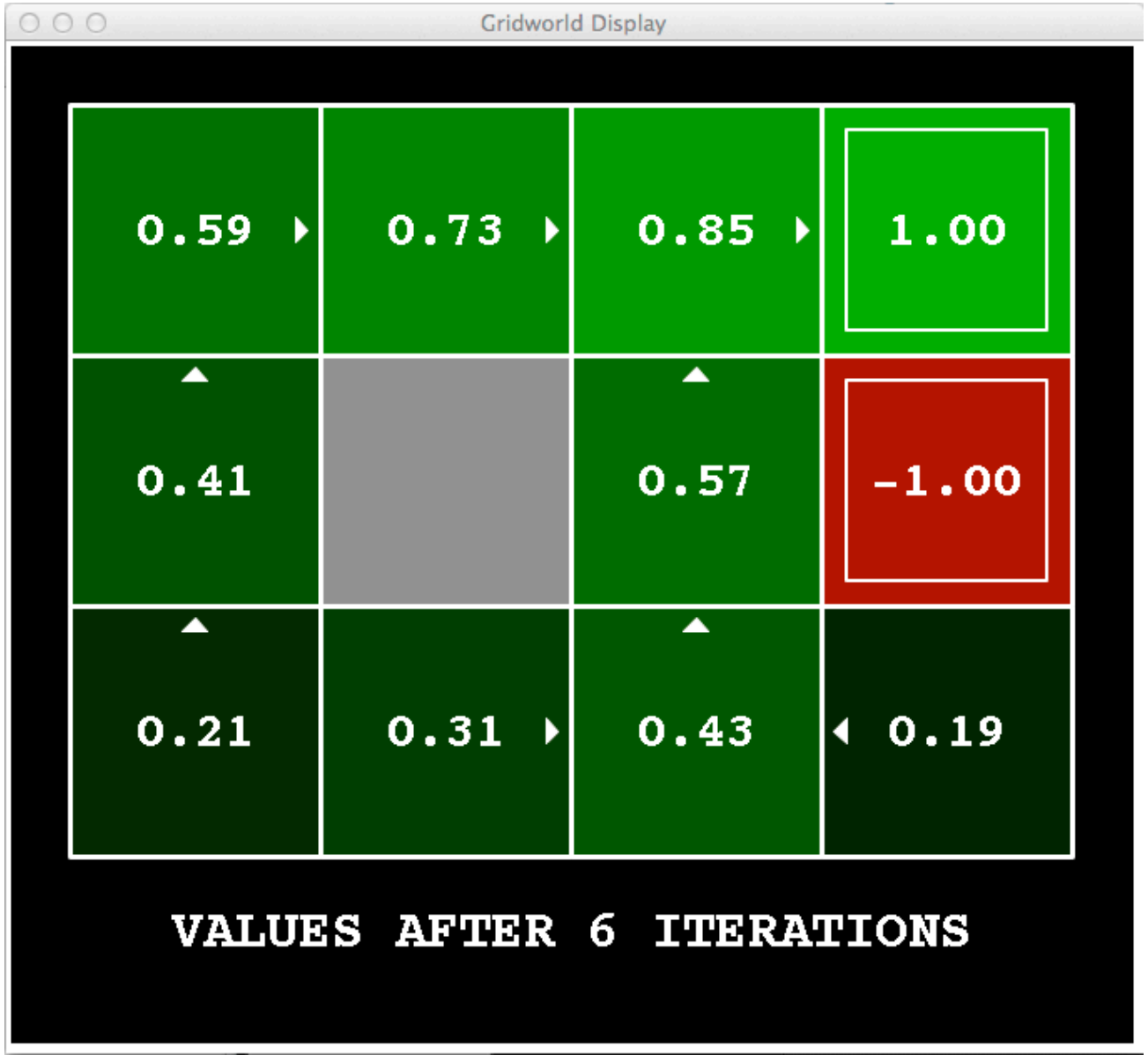
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



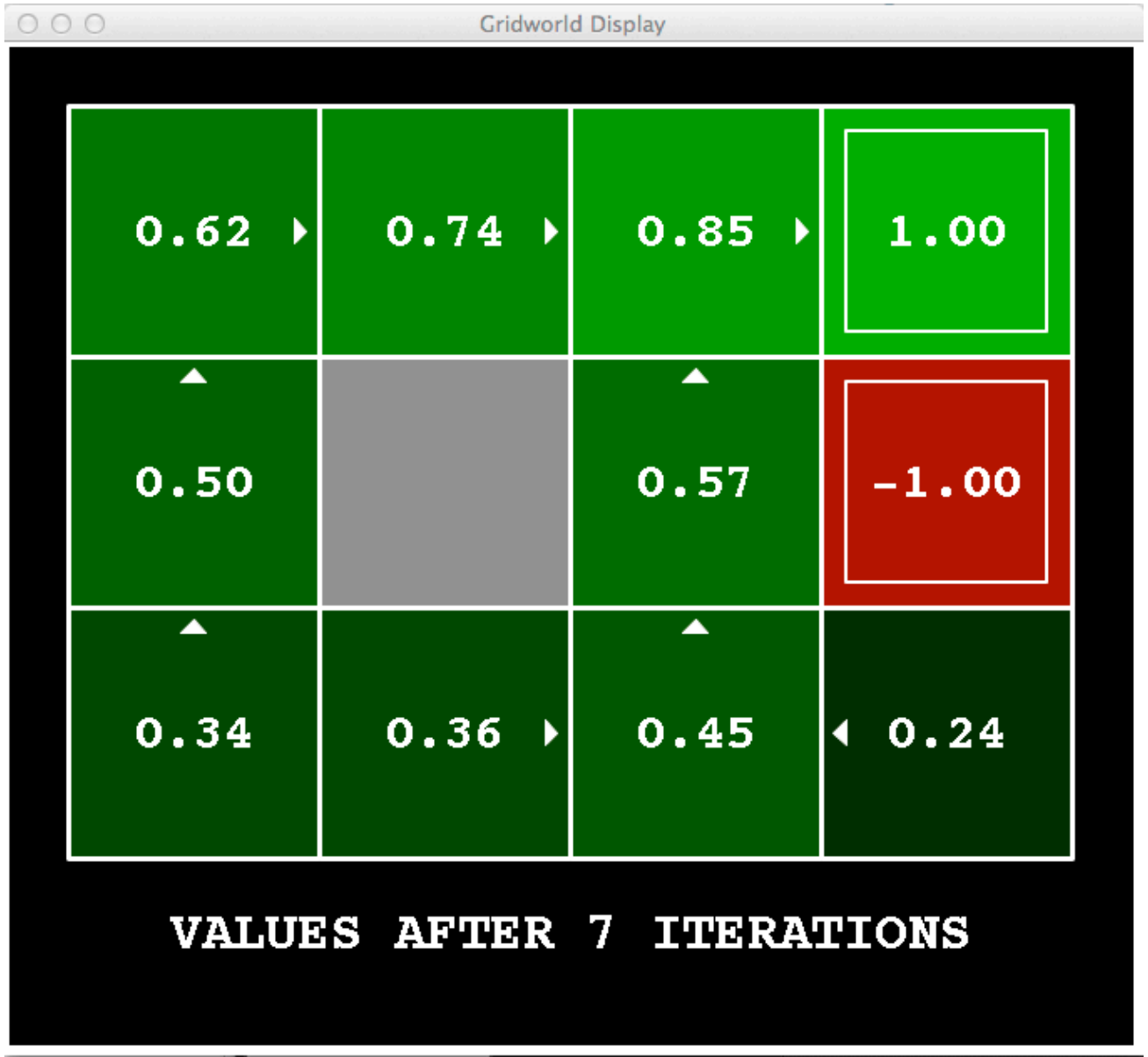
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



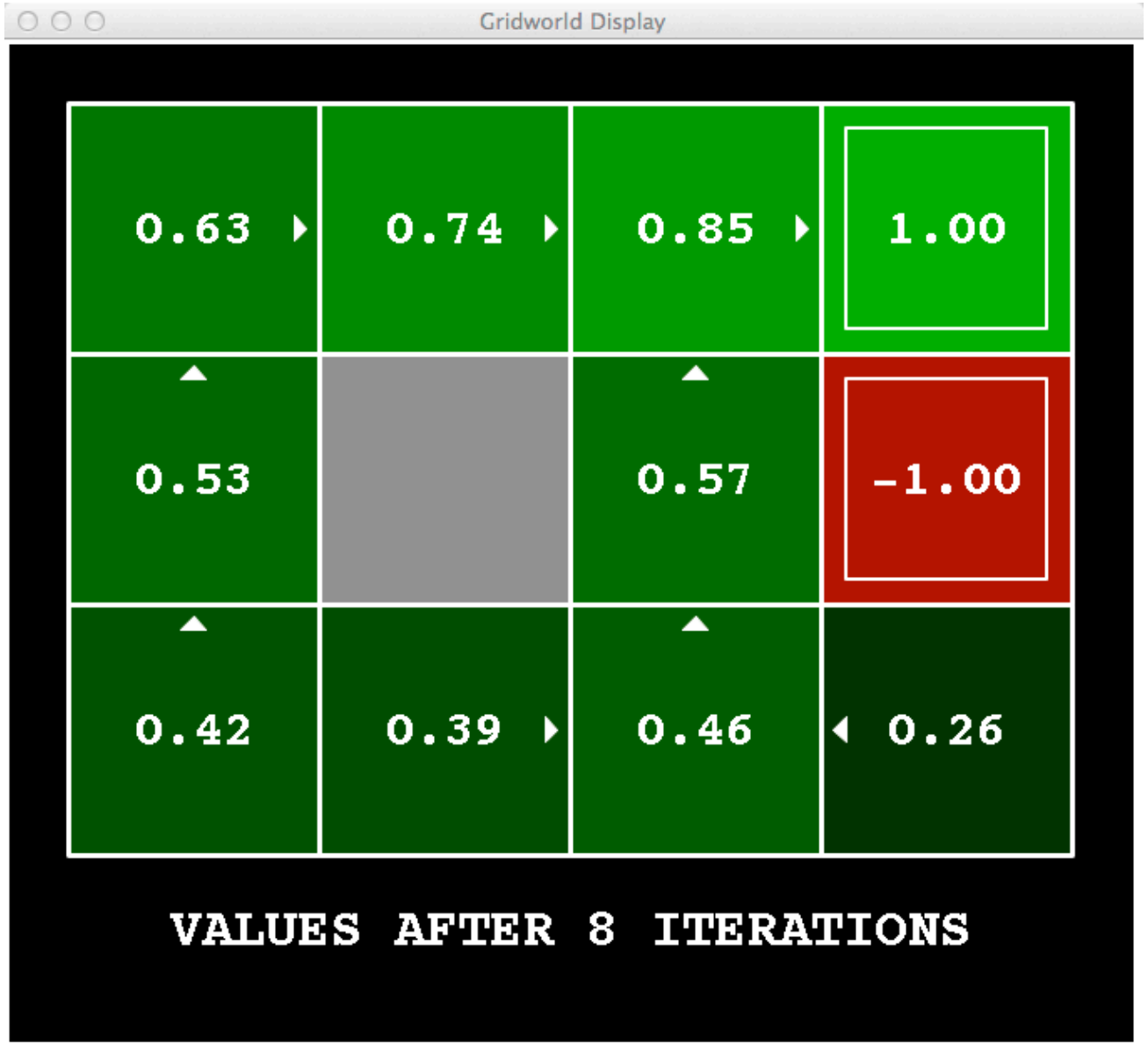
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



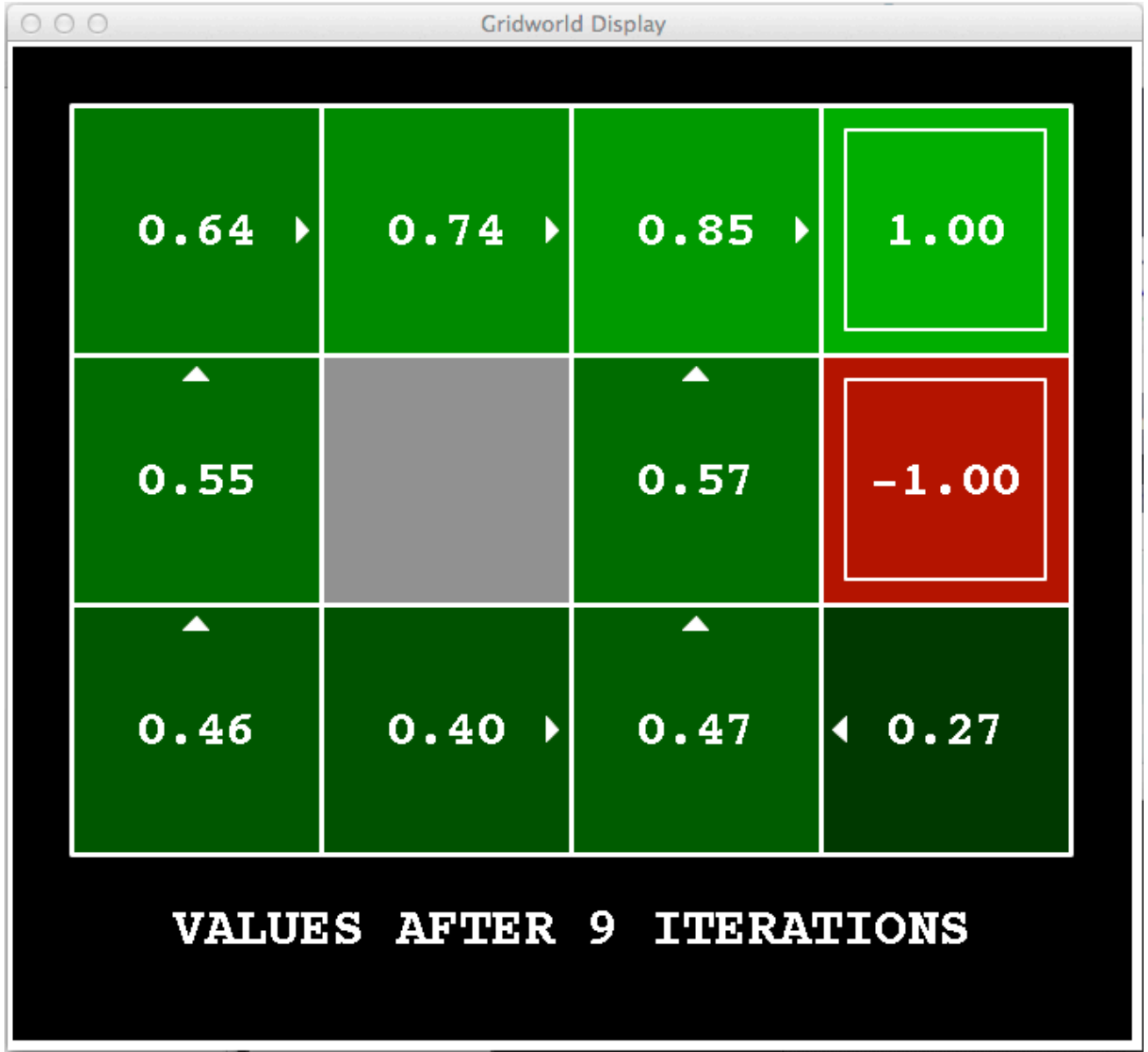
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



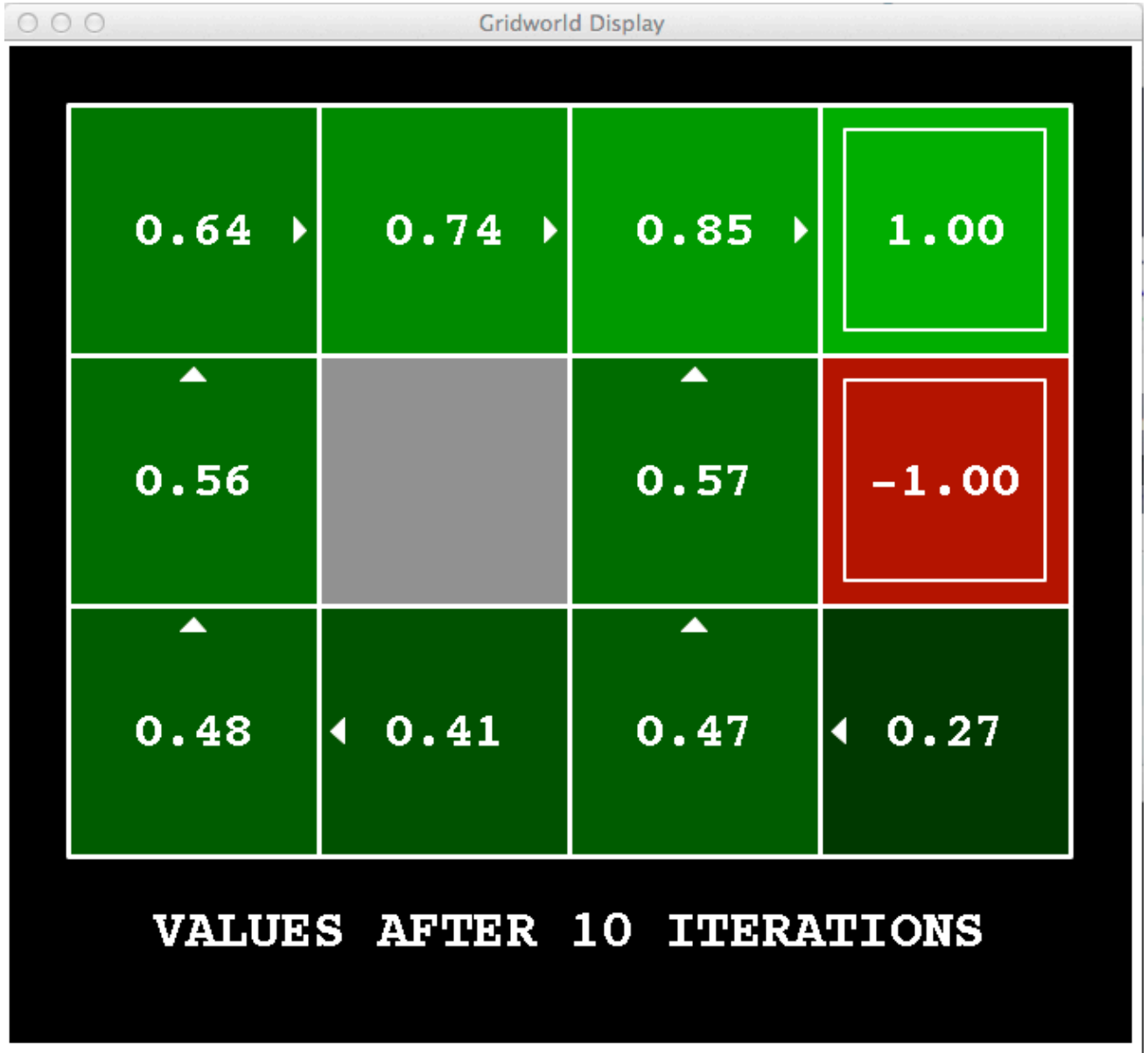
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



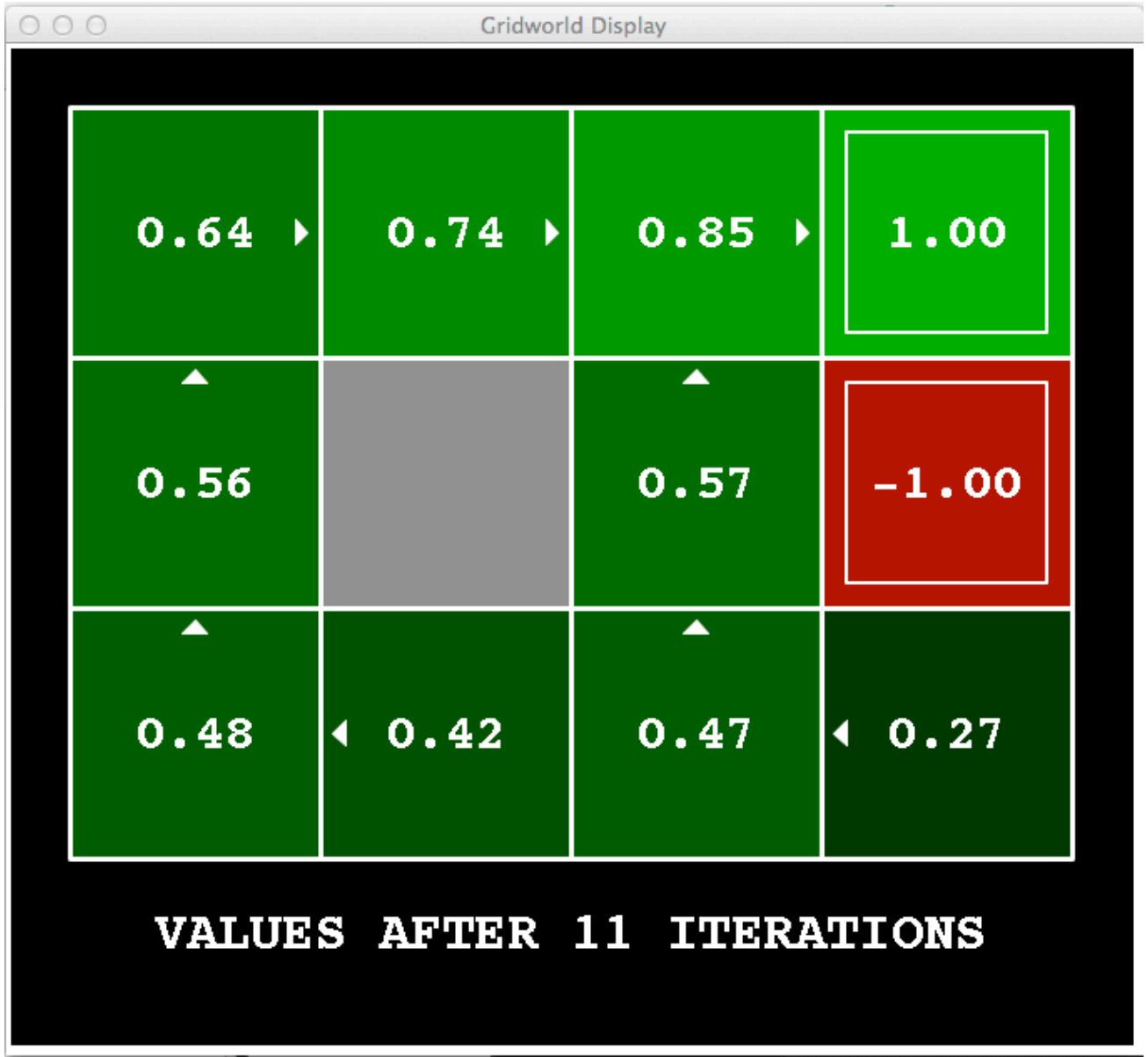
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



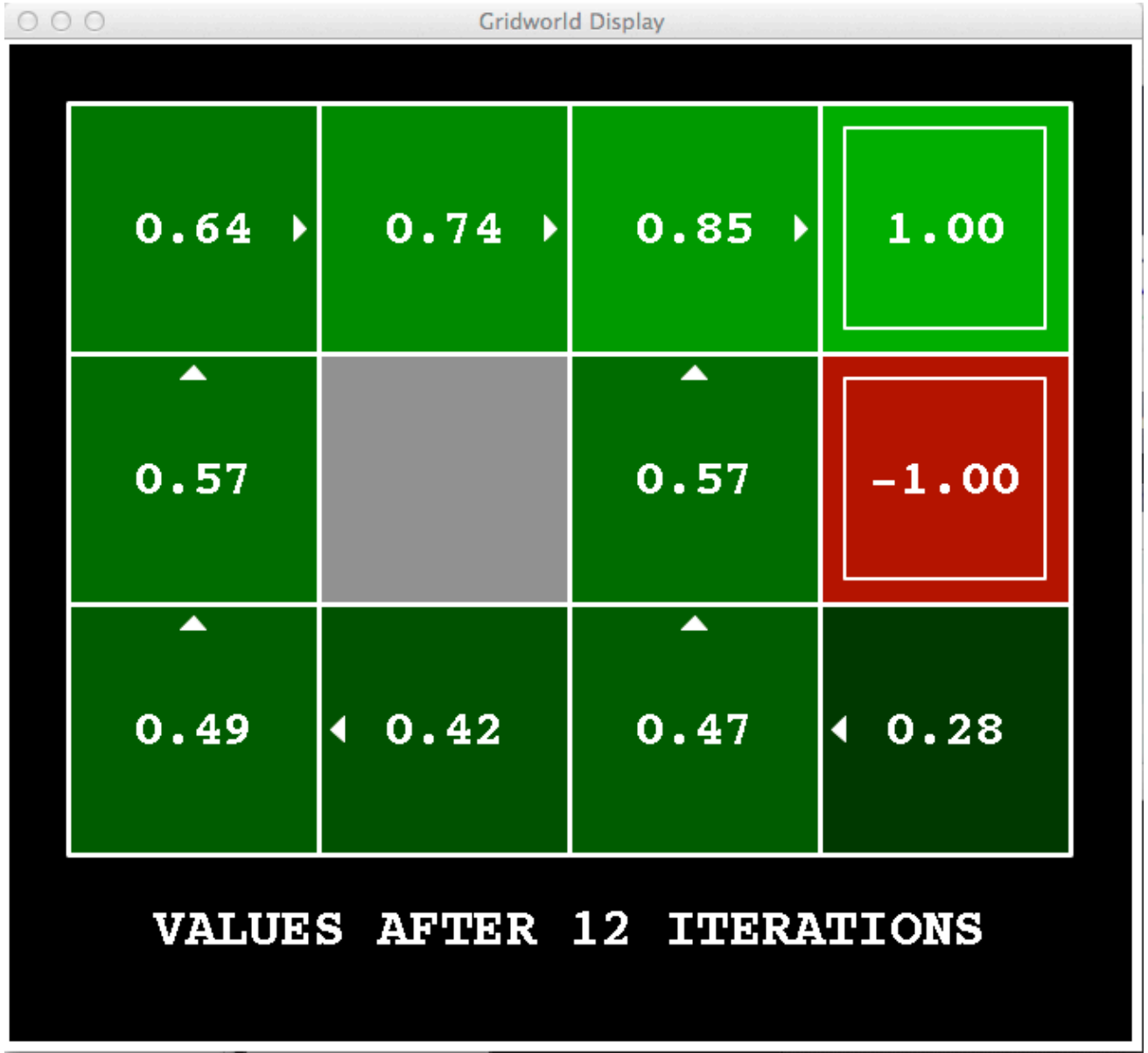
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



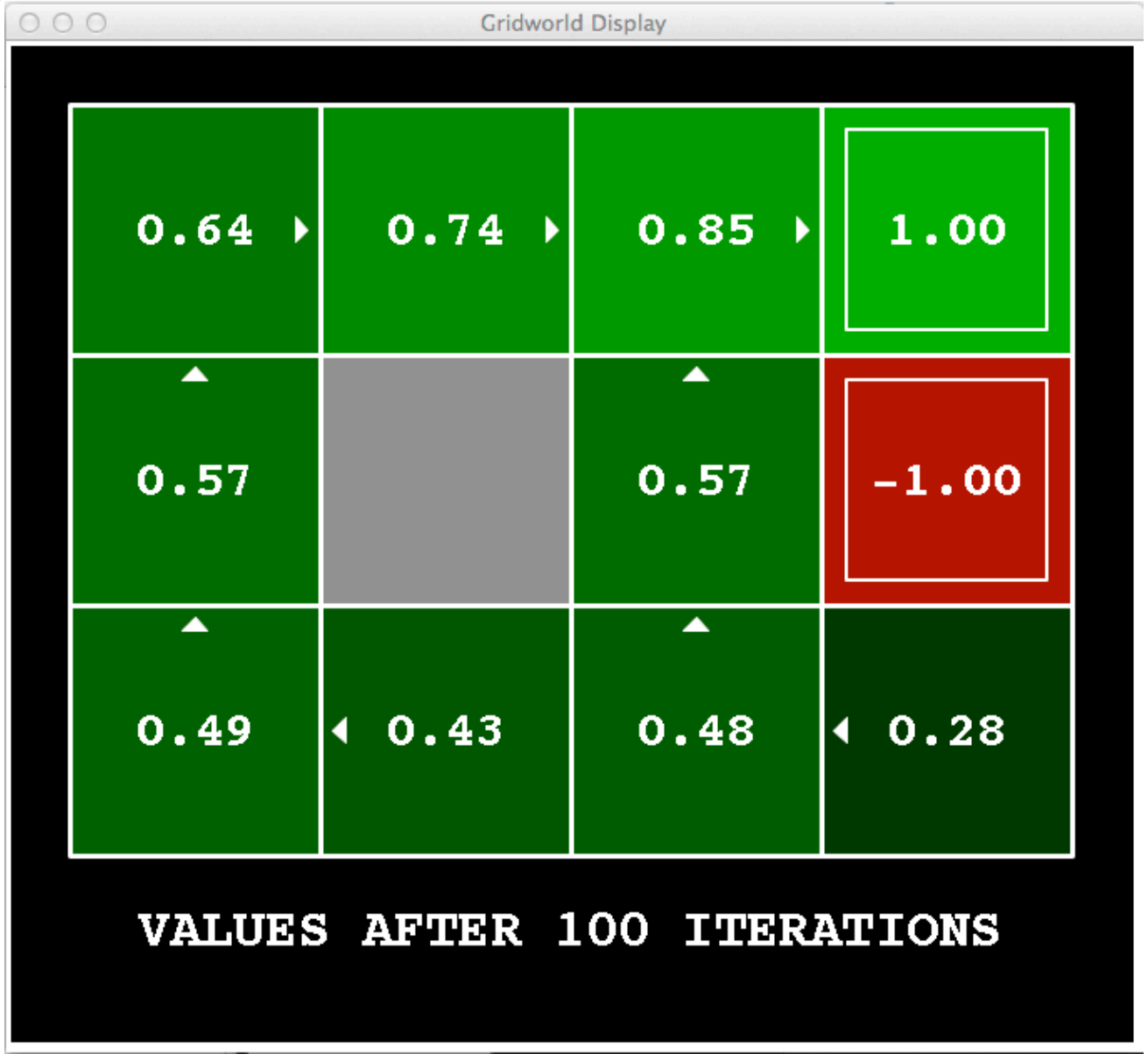
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

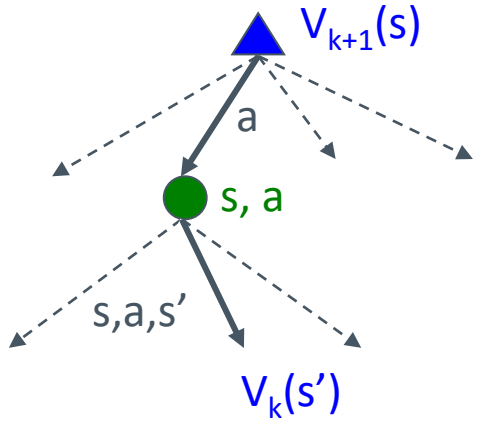


Noise = 0.2
Discount = 0.9
Living reward = 0

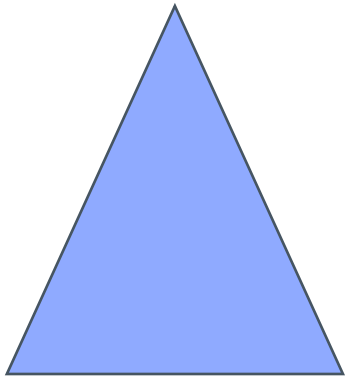
Value Iteration

- 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 step 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
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do



Value iteration

Bellman equations characterize the optimal values:

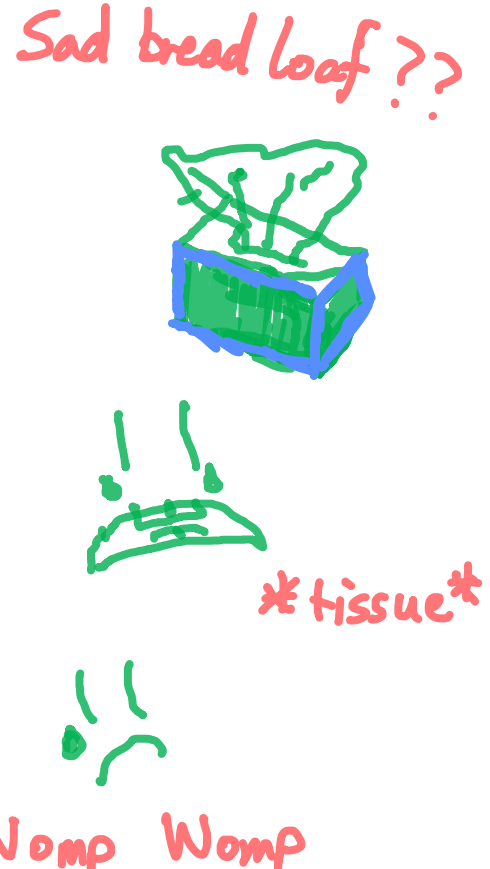
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

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

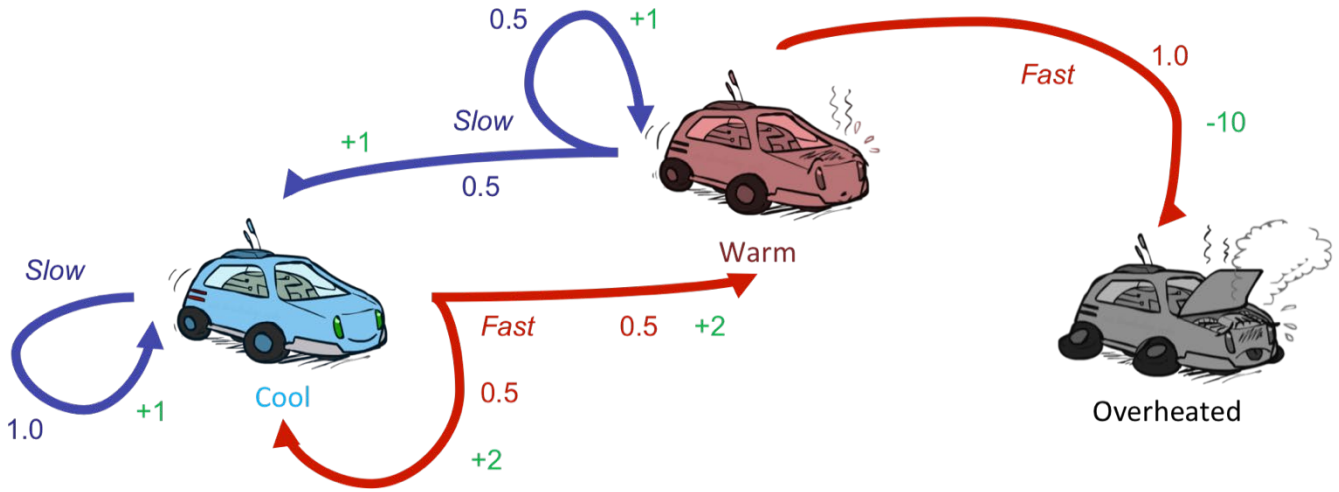


Example: Value Iteration

V_2

V_1

V_0

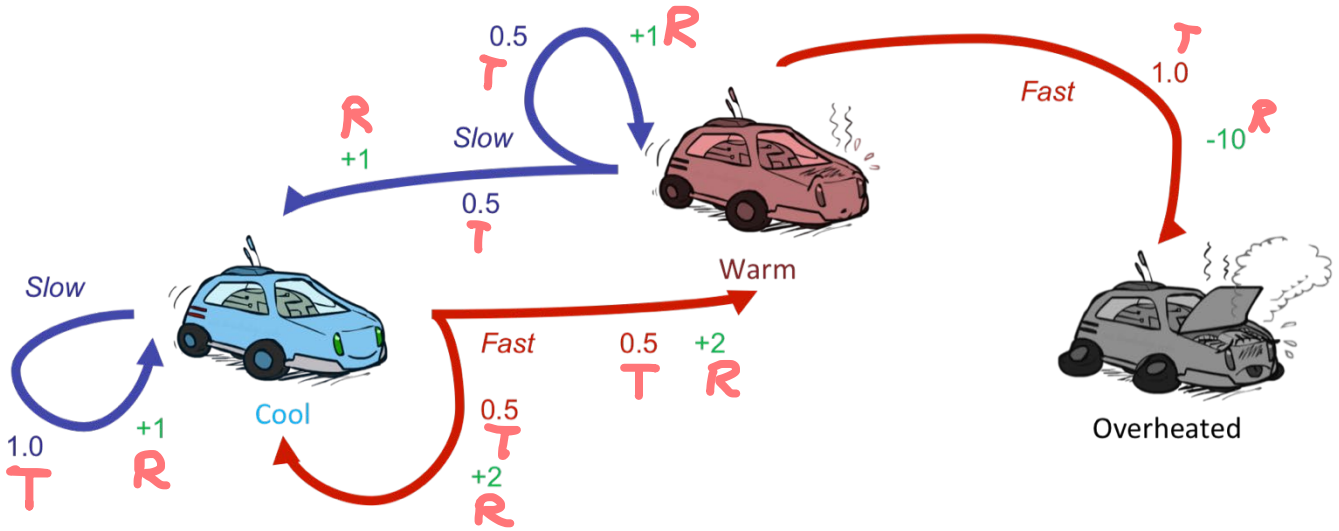


Assume no discount!

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

Example: Value Iteration

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



Assume no discount!

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

max(Action Slow, Action Fast) from a given state

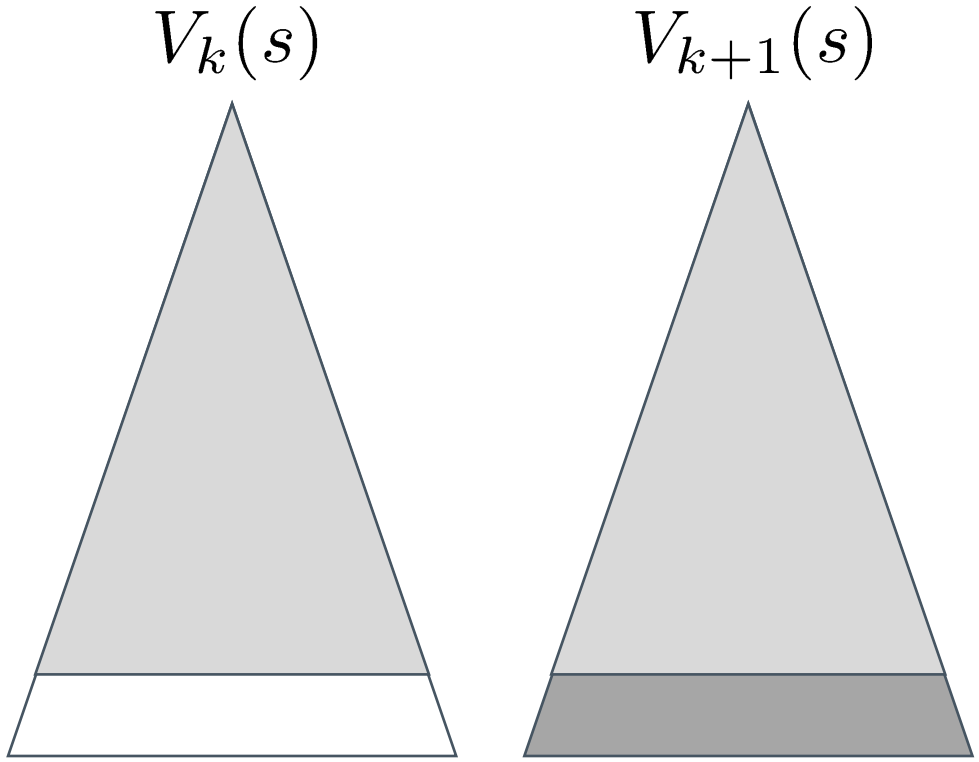
Exercise

› For the following MDP, with discount factor of 1, compute values for V^*_0, V^*_1, V^*_2 for each state A, B

s	a	s'	T(s,a,s')	R(s,a,s')
A	1	A	0	0
A	1	B	1	0
A	2	A	0	0
A	2	B	1	2
A	3	A	0.5	0
A	3	B	0.5	0
B	1	A	0.4	0
B	1	B	0.6	10
B	2	A	0	0
B	2	B	1	0
B	3	A	0.5	2
B	3	B	0.5	6

Convergence*

- 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



Policy iteration

Iterate over Policies

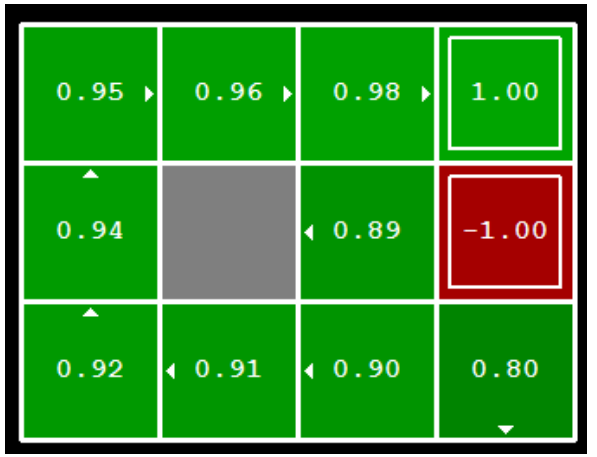
What policy is best for the current state?

- Policy Extraction
- Policy Evaluation
- Policy Iteration
- Policy Improvement
- Value vs Policy Iteration
- RL Value-based vs Policy-Based methods

Policy extraction

Computing Actions from Values

- Let's imagine we have the optimal values V*(s)
- How should we act?
  - It's not obvious!



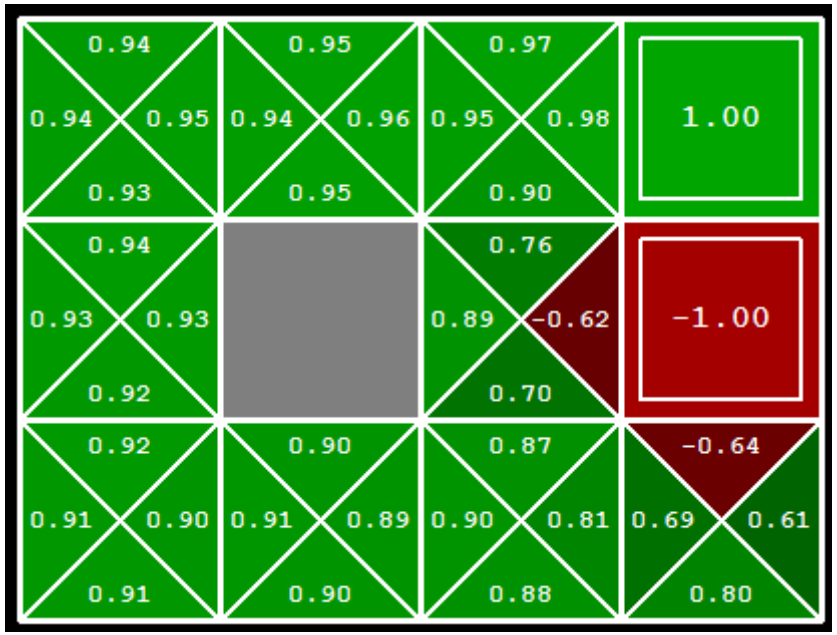
- one step value iteration (expectimax) – compute q-value

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

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

Computing Actions from Q-Values

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



- How should we act?
  - Completely trivial to decide!

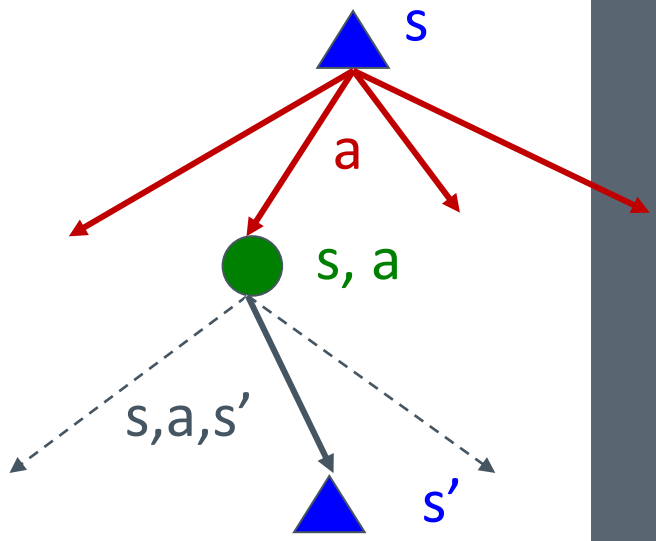
$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

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

Problems with value iteration

› Value iteration repeats the Bellman updates:

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

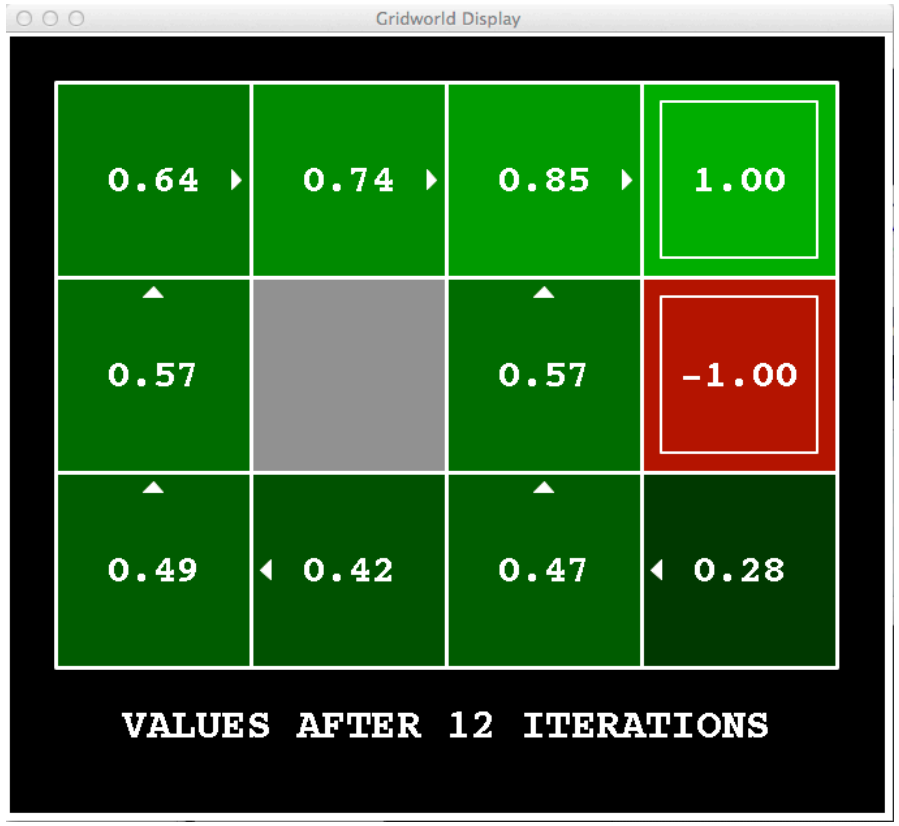


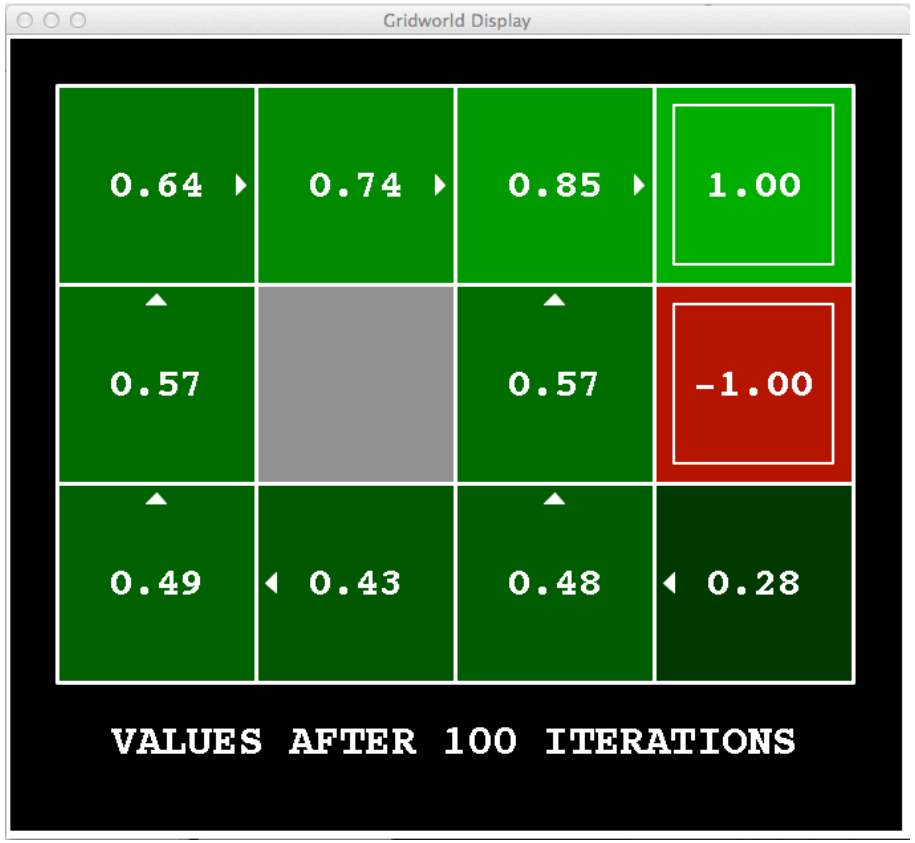
› Problem 1: It's slow – $O(S^2A)$ per iteration

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

› Problem 3: The policy often converges long before the values

Convergence example





How to find optimal policy?

- In order to figure out the optimal policy, it should not be necessary to compute the optimal value function exactly
- Since there are only finitely many policies in a finite-state, finite-action MDP, it is reasonable to expect that a search over policies should terminate in a finite number of steps

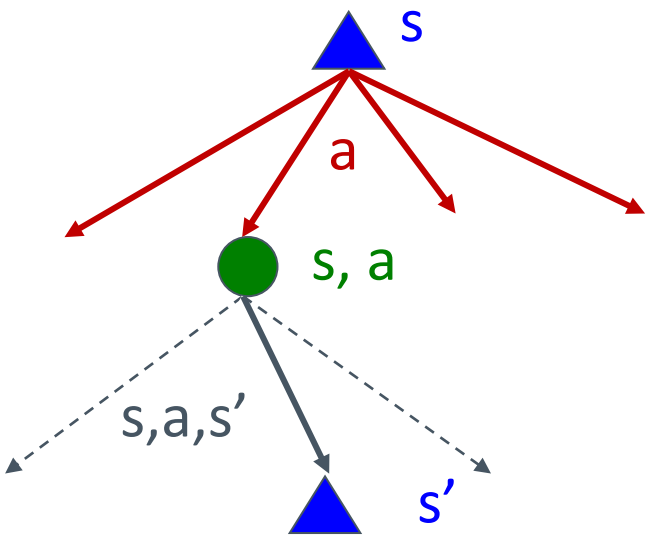
Policy Iteration

- 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 the 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

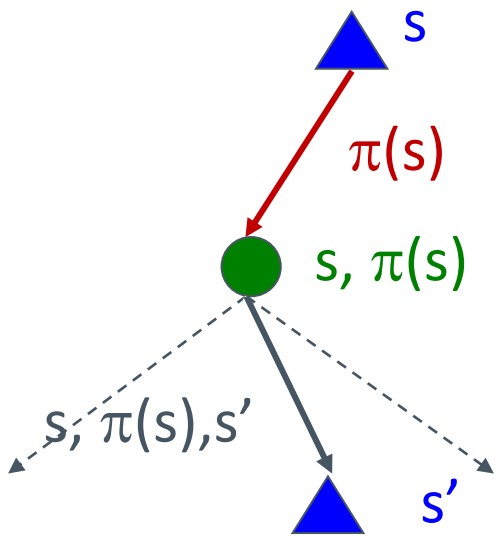
Policy evaluation

Fixed Policies

Do the optimal action



Do what π says to do



- Value iteration - max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state
  - … though the tree's value would depend on which policy we fixed

Utilities for a Fixed Policy

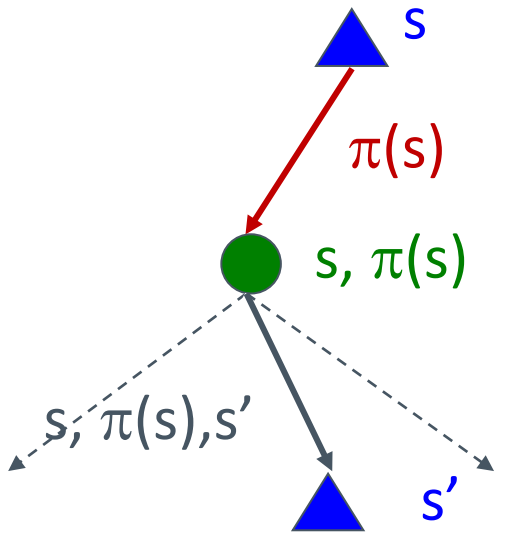
- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- Define the utility of a state s, under a fixed policy π:

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

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

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



Example: Policy Evaluation

Always Go Right

Always Go Forward

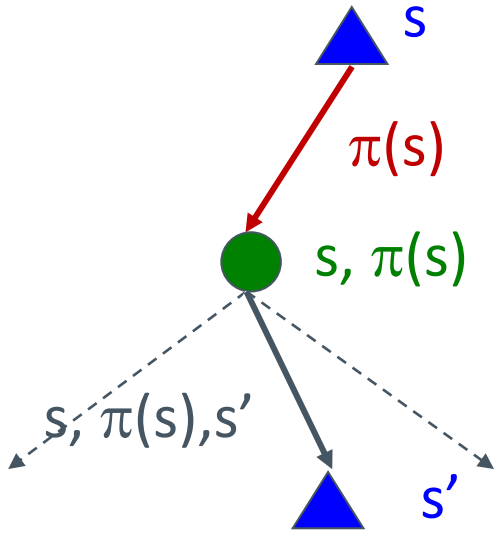
Policy Evaluation

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

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

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

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

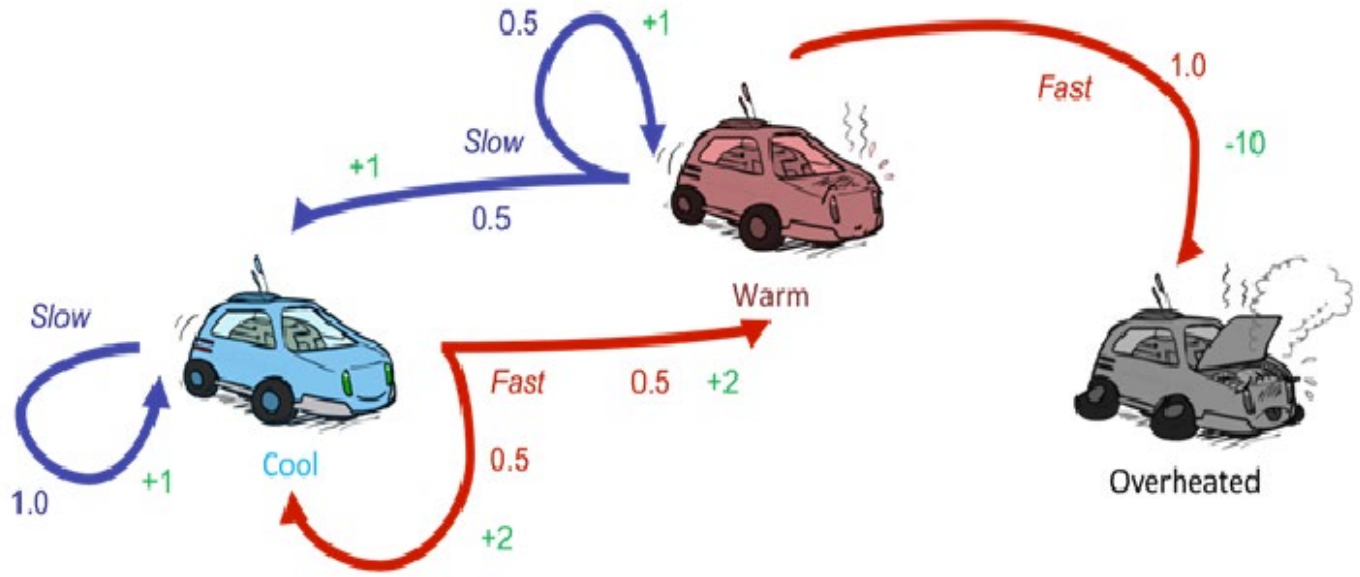


- Efficiency: $O(S^2)$ per iteration

- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)

Policy Evaluation exercise

Evaluate the following policies: always go fast, always go slow
Assume discount of 0.5



Policy improvement

How to improve a policy?

1. Evaluate a given policy
2. Improve the policy by acting greedily with respect to V_π

$$\pi' = \text{greedy}(v_\pi)$$

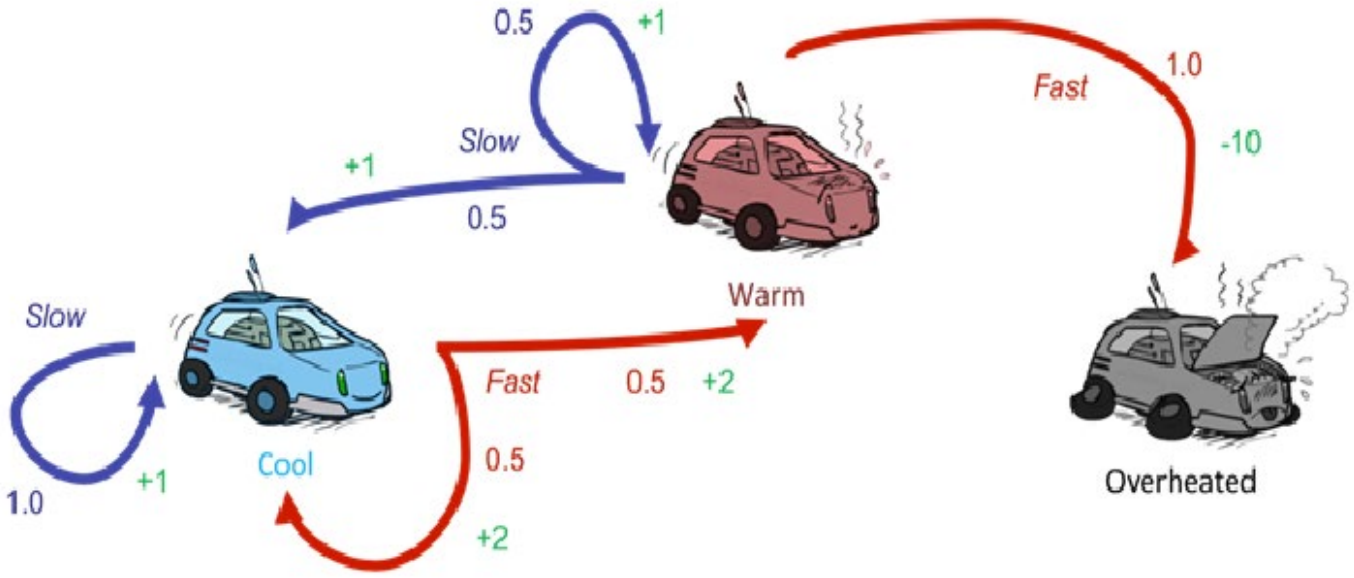
How?

- One-step look-ahead, using policy extraction

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

Policy Improvement exercise

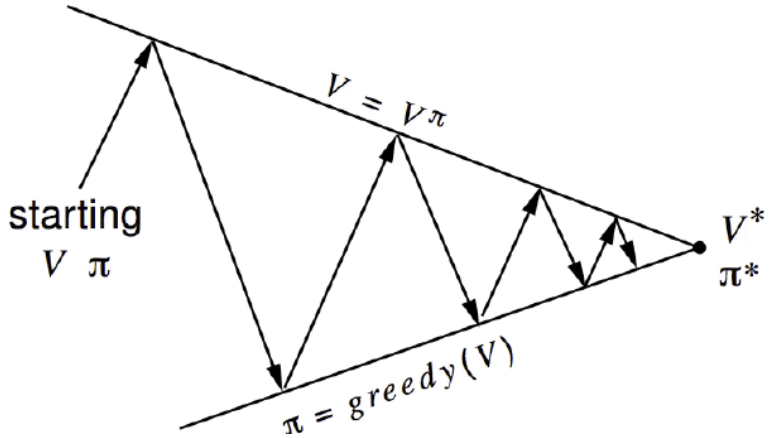
Do one-step improvement of “always go slow” and “always go fast” policies

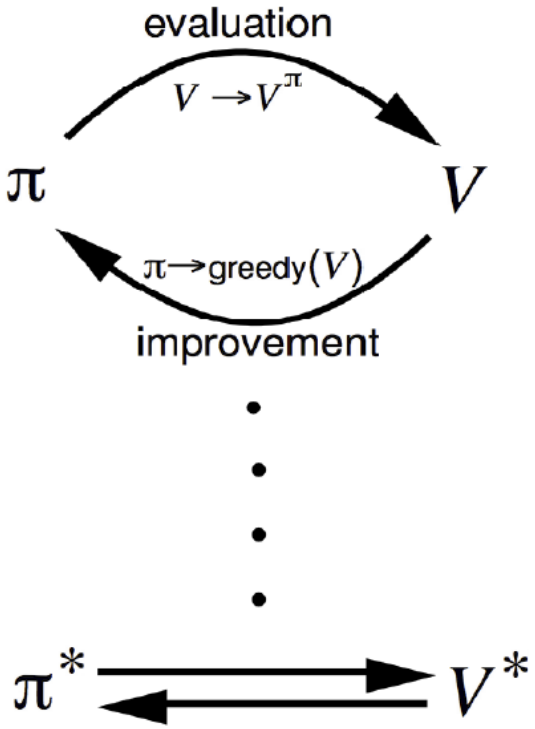


Policy iteration

Policy Iteration

- Iterate until policy converges (remains unchanged)
  - Policy Evaluation
  - Policy Improvement





Policy Iteration

1. Pick an arbitrary policy
2. Iterate
   1. Policy evaluation – solving the linear system of equations for all s

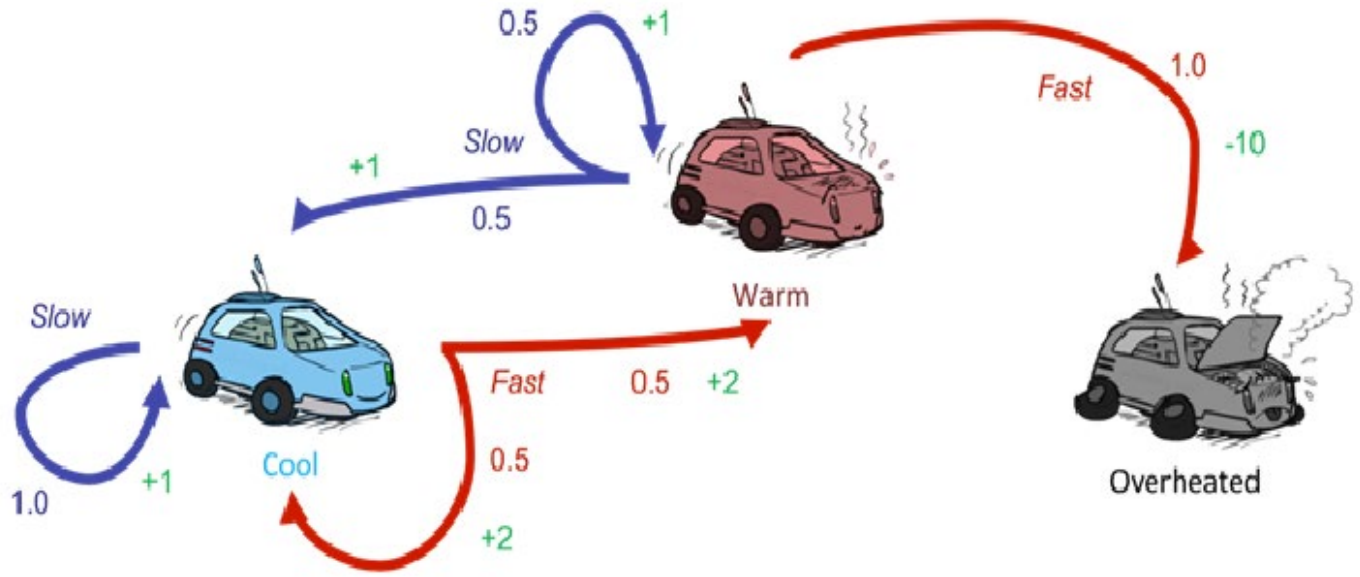
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

   2. Policy improvement for all s

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

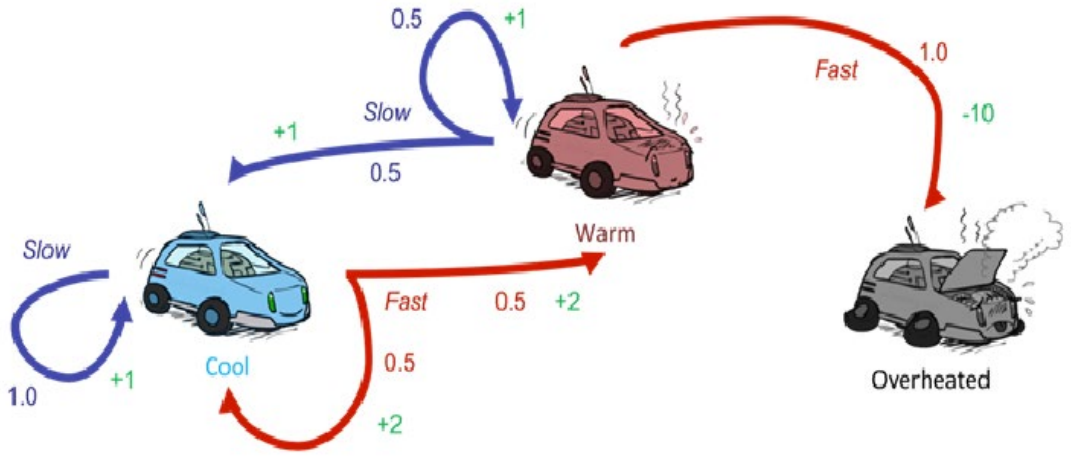
Policy Iteration exercise

Do one-step improvement of “always go slow” and “always go fast” policies
Discount = 0.5



Policy Iteration exercise

Iterate over initial “always go slow” and “always go fast” policies until they converge

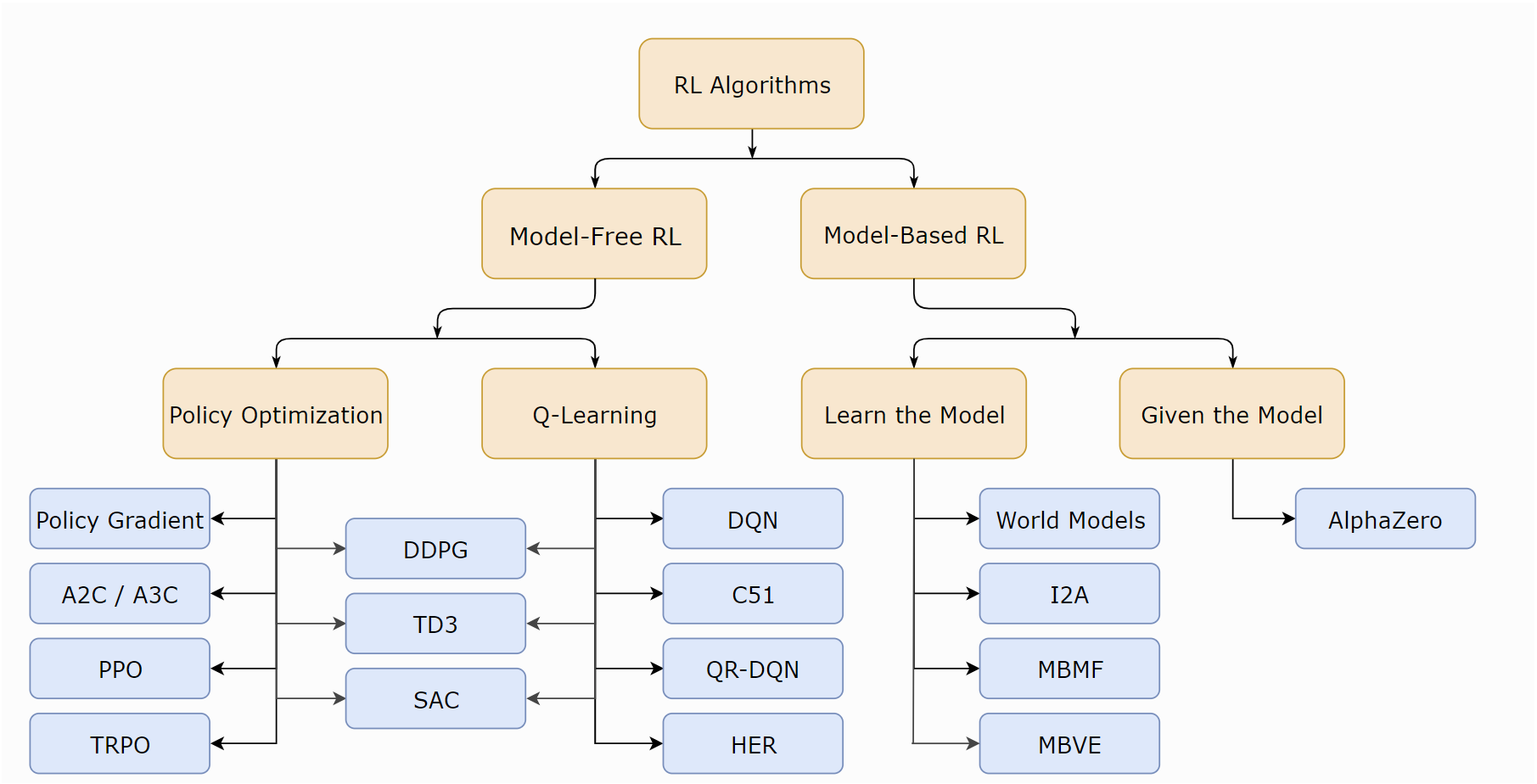


Did both starting policies converge to the same final optimal policy? How does this compare to policy obtained by value iteration in the last lecture?

Summary – Value-iteration vs Policy-iteration

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)
- Both are dynamic programs for solving MDPs
- Assume **full knowledge of the MDP**
- Both used for **offline planning** in an MDP

Value-based vs Policy-based methods in RL



https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Value-iteration vs Policy-iteration in RL

- Which to use?
- Depends on the problem – state space might lends itself better to one or the other
  - Eg large and continuous state space -> policy methods
- Value-based – simple, fast – not always stable
- Combining the two – actor-critic methods

Implementation examples

- Credit: https://github.com/tirthajyoti/RL_basics
- MDP_VI_PI_Q-learning_AIMA.ipynb
  - Value-iteration visualisation
  - Policy iteration on 2 grids
  - Tests for computational efficiency of both, based on grid size
- https://github.com/dennybritz/reinforcement-learning/tree/master/DP

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