Freie Universität Berlin

Lecture: Machine Learning for Data Science

Winter semester 2021/22

Lecture 16: Reinforcement Learning (Intro and MDPs)

Prof. Dr. Eirini Ntoutsi

Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

Main machine learning tasks

fruit	length	width	weight	label
fruit 1	165	38	172	Banana
fruit 2	218	39	230	Banana
fruit 3	76	80	145	Orange
fruit 4	145	35	150	Banana
fruit 5	90	88	160	Orange
...				
fruit n	...	...	...	...

- Based on the feedback we have on the data, we can distinguish between:
- Direct-feedback instances — Supervised learning
  - the correct response /label is provided for each instance by the "teacher"
  - e.g., good or bad product

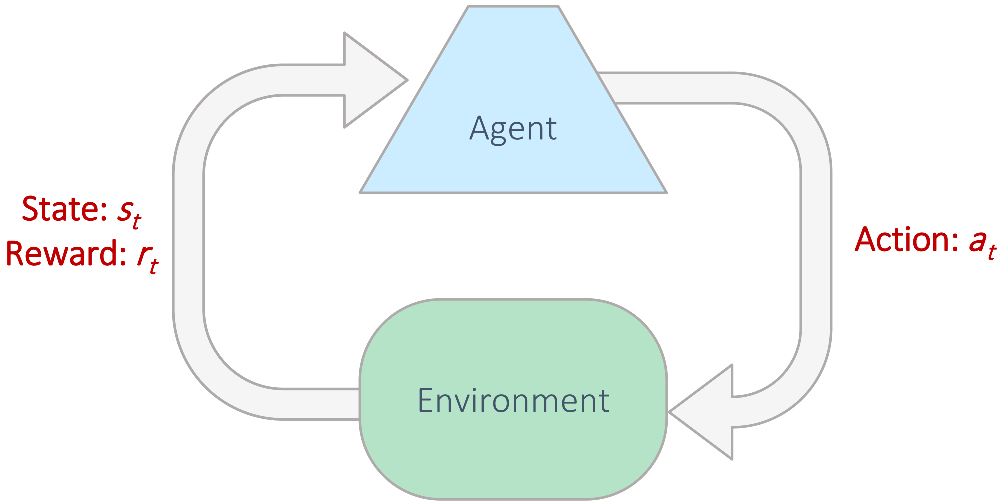
fruit	length	width	weight
fruit 1	165	38	172
fruit 2	218	39	230
fruit 3	76	80	145
fruit 4	145	35	150
fruit 5	90	88	160
...			
fruit n	...	...	...

- No-feedback instances — Unsupervised learning
  - no evaluation/label of the instance is provided, since there is no "teacher"
  - e.g., no information on whether a product is good or bad, just the description of the product/instance

- Indirect-feedback instances — Reinforcement learning
  - less feedback is given, since not the proper action, but only an evaluation of the chosen action is given by the "teacher"

Agent and environment

- RL is a type of ML technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences.



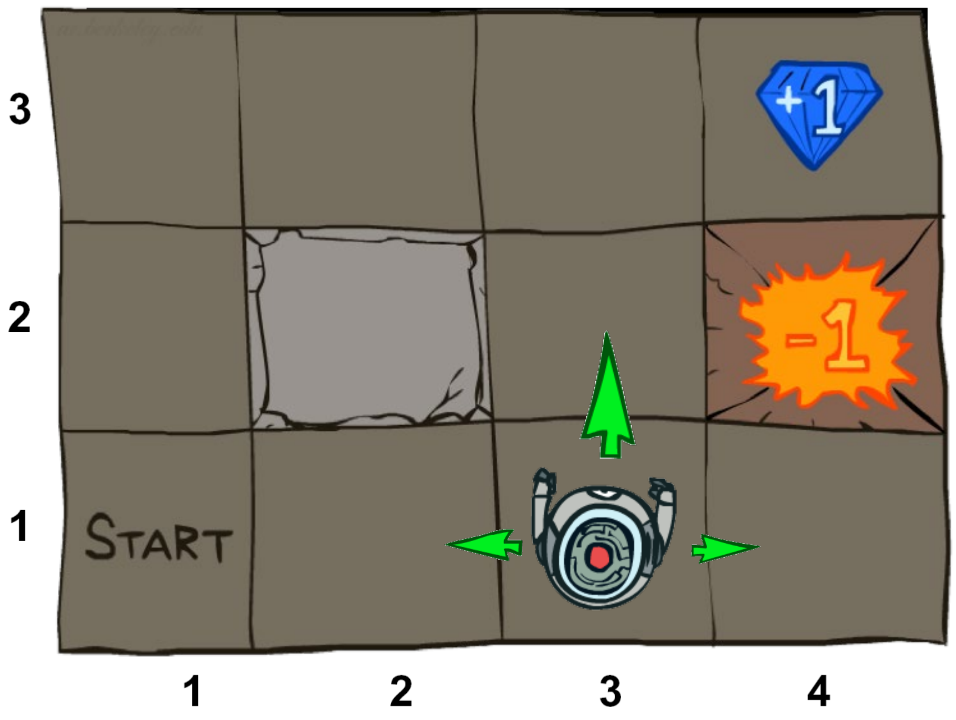
- At each step t
  - The agent executes action a_t
  - Transitions to state s_t
  - Receives scalar reward r_t
    - reward can be sparse

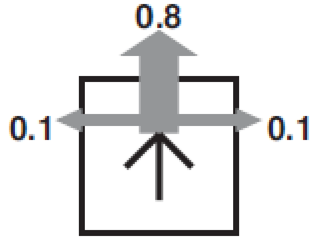
In the terminology of phycology, reward is reinforcement

- The goal of the agent is to learn to choose actions so as to maximize the sum of rewards

Example: Grid World

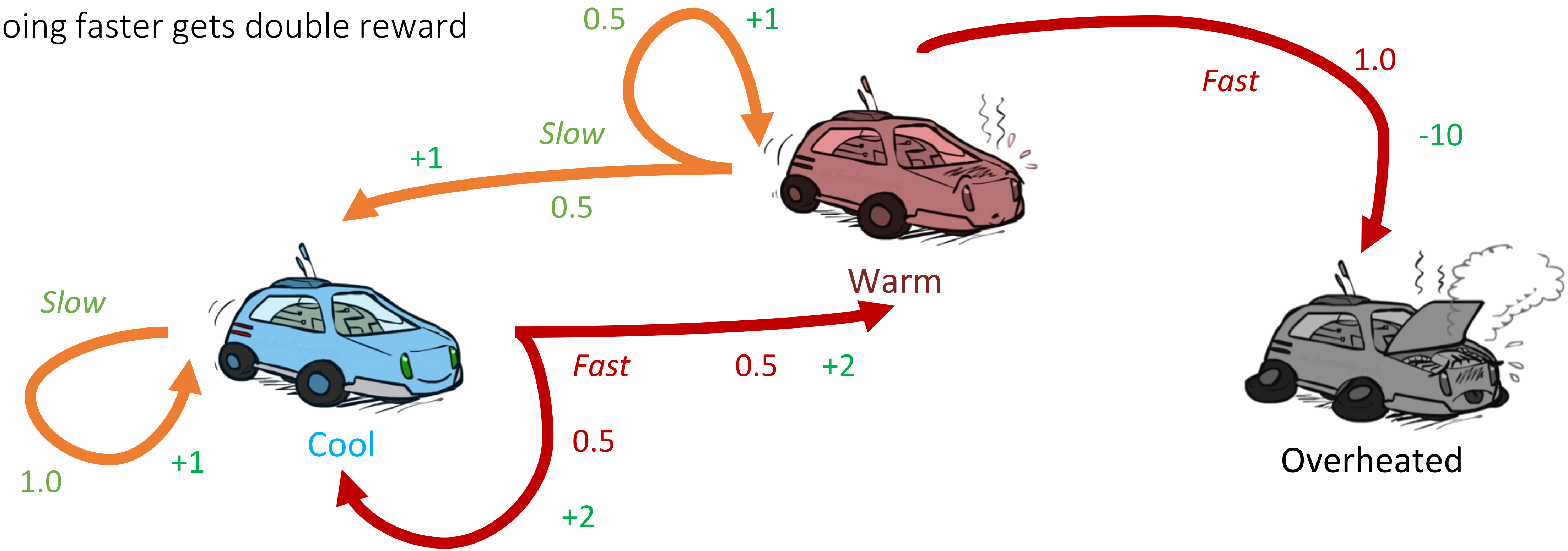
- A maze-like problem
  - The agent lives in a grid, walls block its path
  - States: different positions of the agent in the grid, assuming that wall, diamond, fire do not move
  - Actions: {North, West, East, South}
- The agent receives rewards at each time step
  - Indicates how well agent is doing at step *t*
  - Small "living" reward each step (can be negative), e.g., -0.04
  - Big rewards come at the end (good or bad)
- Uncertainty: actions do not always go as planned
  - E.g., 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
- Goal: maximize sum of rewards





Example: Racing

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

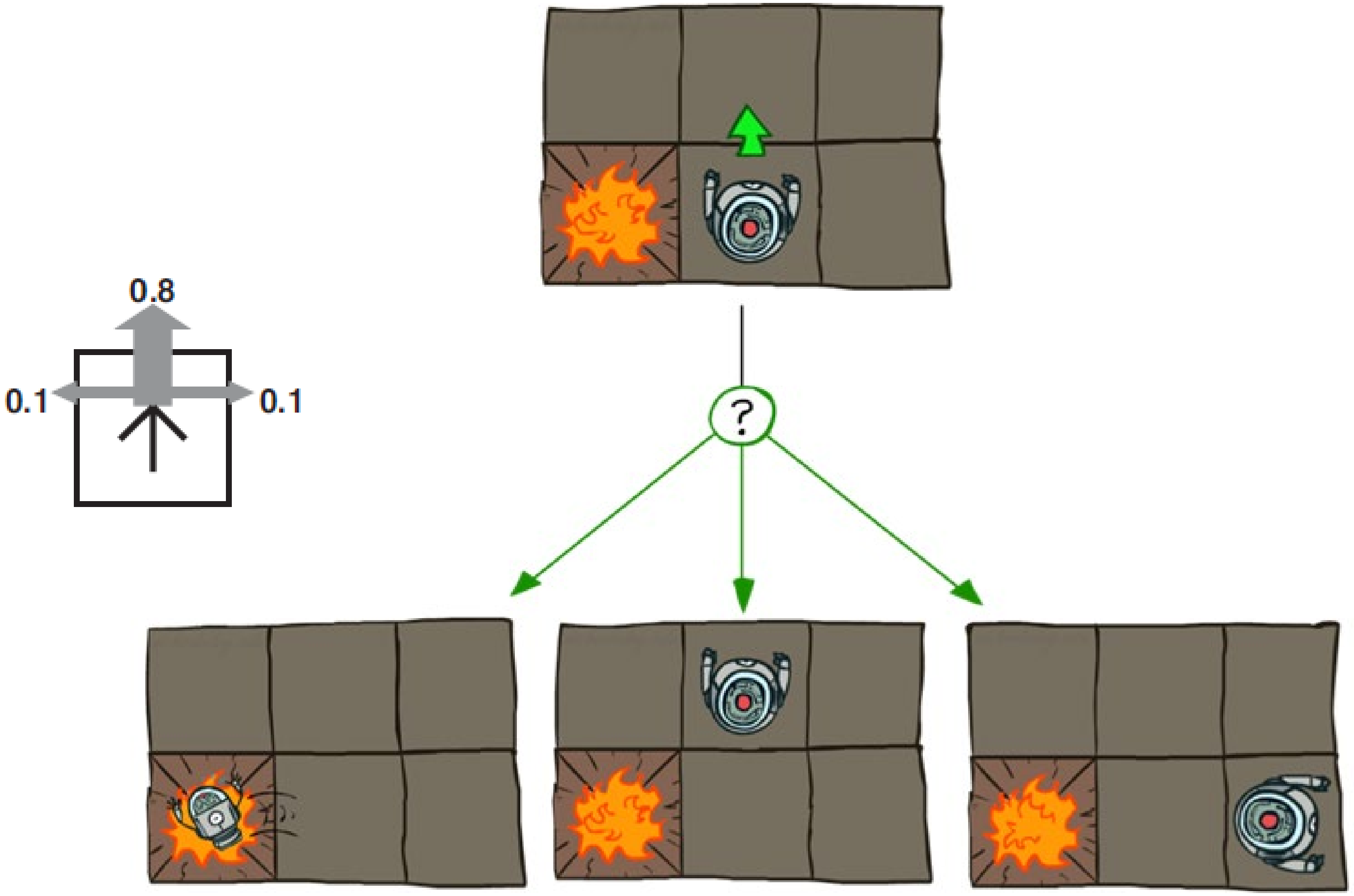


Example applications

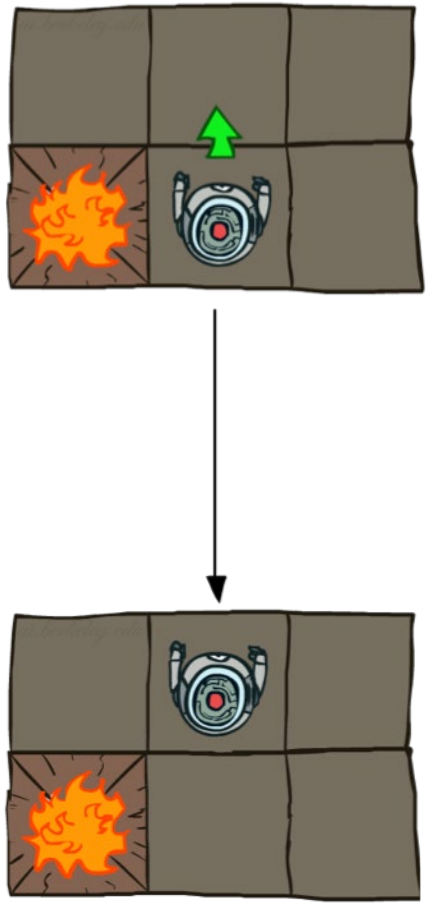
- Play Backgammon
  - +/−ve reward for winning/losing a game
- Manage an investment portfolio
  - +ve reward for each $ in bank
- Make a robot walk
  - +ve reward for forward motion
  - −ve reward for falling over
- Taxi driving
  - +ve reward for getting closer
  - -ve reward for moving away from the destination
- ...

Characteristics of RL: non-deterministic decision making

- Non-deterministic: There is uncertainty associated with the actions
  - Actions might result in multiple successor states

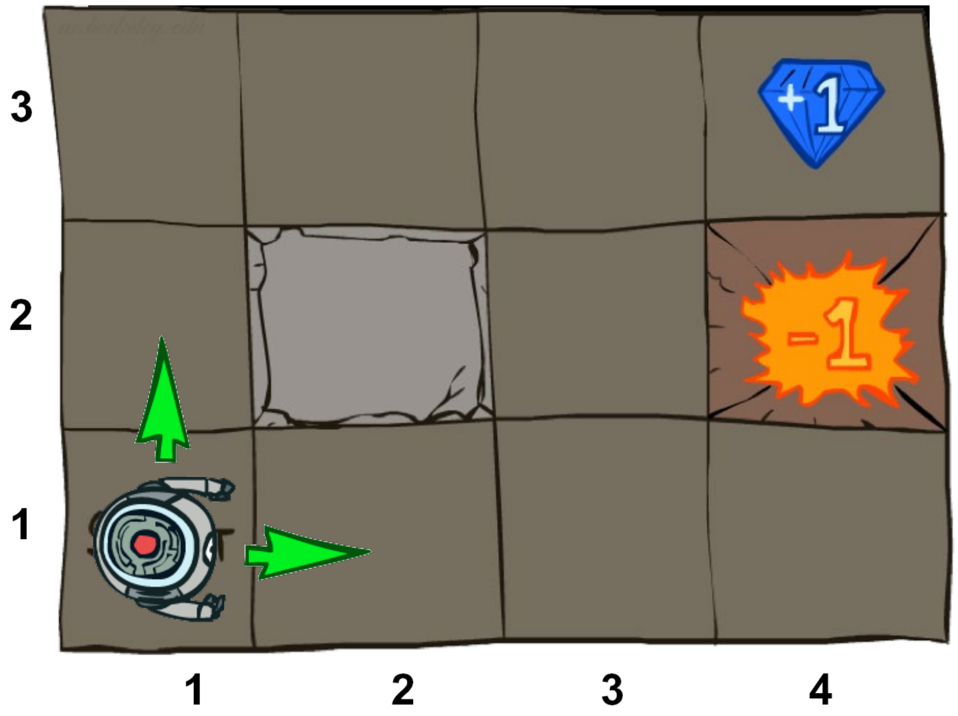


Deterministic grid world



Characteristics of RL: sequential decision making

- Sequential decision making:
  - the current decision could affect all future decisions.
  - short-term actions can have long-term consequences.
- Remember the goal of the agent to is to maximize the sum of rewards
  - It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
  - A financial investment (may take months to mature)
  - Refuelling a helicopter (might prevent a crash in several hours)
  - Blocking opponent moves (might help winning chances many moves from now)



What makes RL different from other ML tasks?

- Agent receives feedback in the form of rewards
  - There is no supervisor/teacher, only a reward signal
- Feedback is delayed, not instantaneous
- Time really matters (sequential, non i.i.d data)
- Agent's actions affect the subsequent data it receives

How the agent learns?

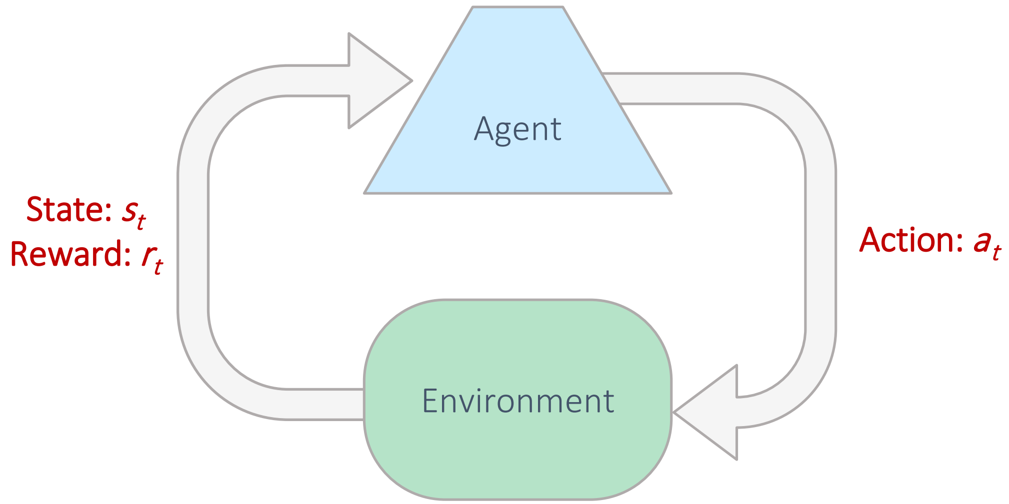
- The goal of the agent is to learn to choose actions so as to maximize the sum of rewards
- How the agent learns?
  - By trying out actions and observing the outcomes ← data
    - More on this later

Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

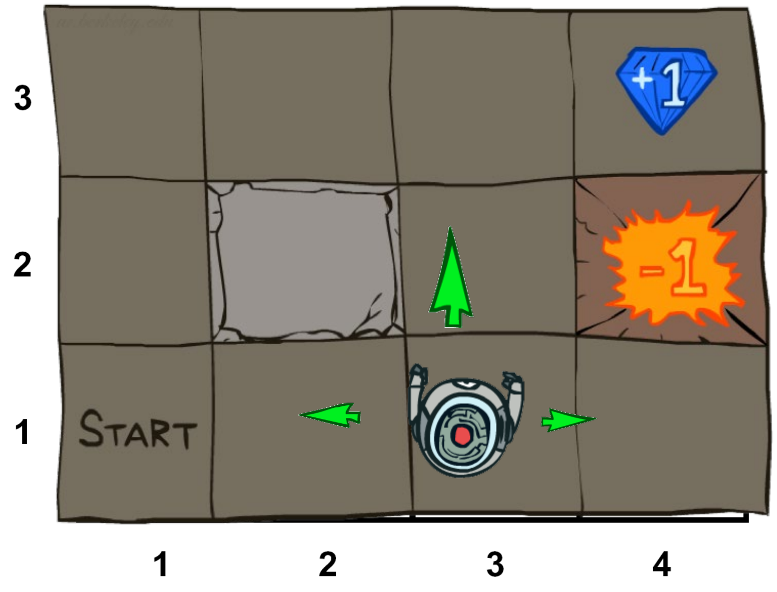
Markov Decision Process (MDP) formulation

- The interaction of the agent with the environment is modeled as a Markov Decision Process (MDP)



Markov Decision Problem (MDP) formulation

For the moment, we assume a full model of the world - We will relax this definition later



- A (full) MDP is defined by:
  - A set of states $s \in S$
    - S_0 the start/initial state
    - Maybe a terminal state
  - A set of actions $a \in A$
    - *Actions(s)*: available actions in state *s*
  - A transition model *T(s, a, s')*: *T(s, a, s')* is the probability that state *s'* is reached, if action *a* is executed in state *s*.
    - Sometimes also, as P(s'|s,a)
    - Transitions are Markovian
  - Reward function *R(s, a, s')* : At each step the agent receives a reward
    - Small living reward, higher reward (bad, good) at the terminal states
    - Sometimes just *R(s)* or *R(s')*

What is Markov about MDPs?

- "Markov" generally means "The future is independent of the past given the present"
- For Markov decision processes (MDPs), "Markov" means action outcomes depend only on the current state

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

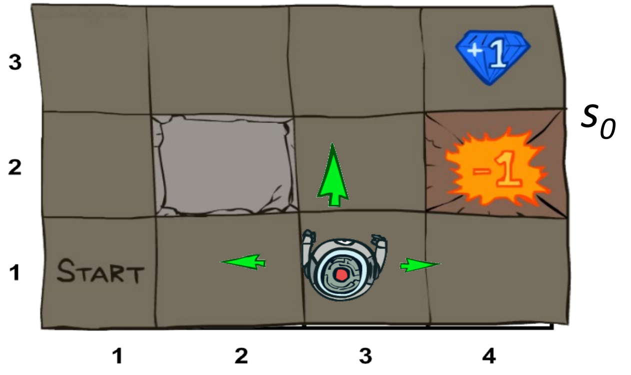
$$=$$

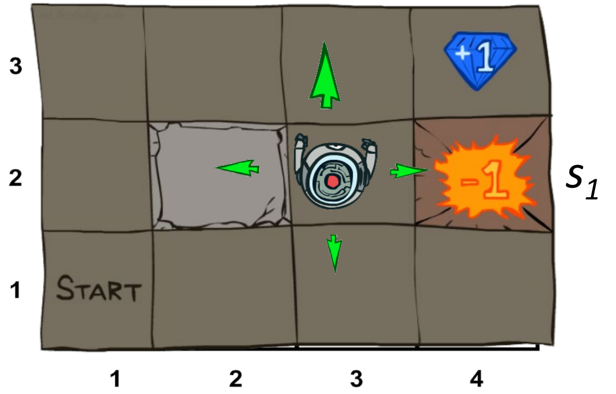
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

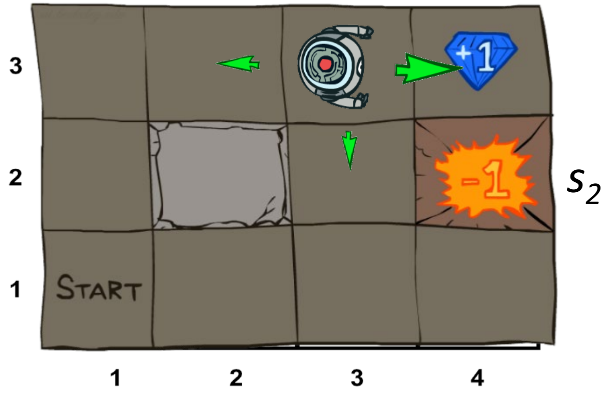
Andrey Markov
(1856-1922)

Environment history and utility of the agent

- Environment history: the sequence of states [s_0,s_1,…, s_n] "experienced" by the agent
- The utility of the agent depends on the environment history!
  - Utility = sum of rewards
- The term utility comes from AI (utility-based agents act based not only goals but also the best way to achieve the goal)
  - Utility-based agents help to choose the best alternatives, when there are multiple alternatives available.







What sort of solutions we are looking for?

- Given the many choices there are many possible solutions
- A solution must specify what the agent should do for any state that the agent might reach → policy
- A policy π*: S→A* defines the agent's behavior for each state $\pi(s)$
  - Deterministic policy: $\pi(s)=a$
  - Non-deterministic/Stochastic policy: $\pi(s)$ is a distribution over possible actions given s_i

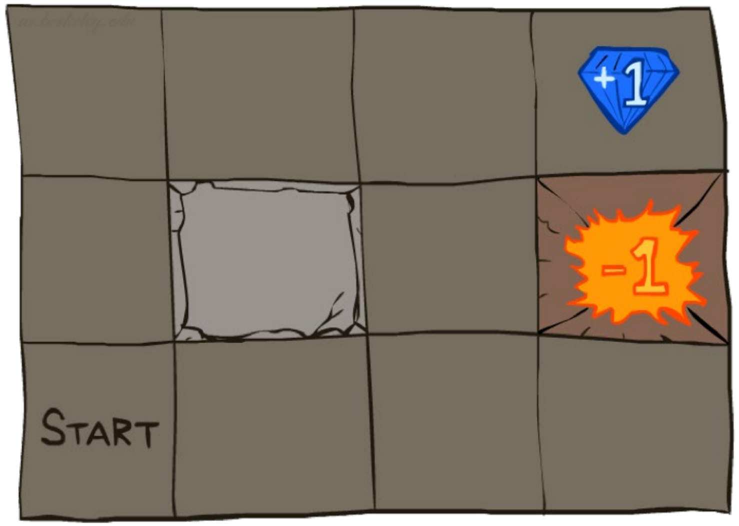
- How can we evaluate a policy π?
  - Simple answer: By checking its utility (=sum of rewards)
  - But each time a policy π is executed starting from S_0, the stochastic nature of the environment might lead to a different environment history generated by the policy:

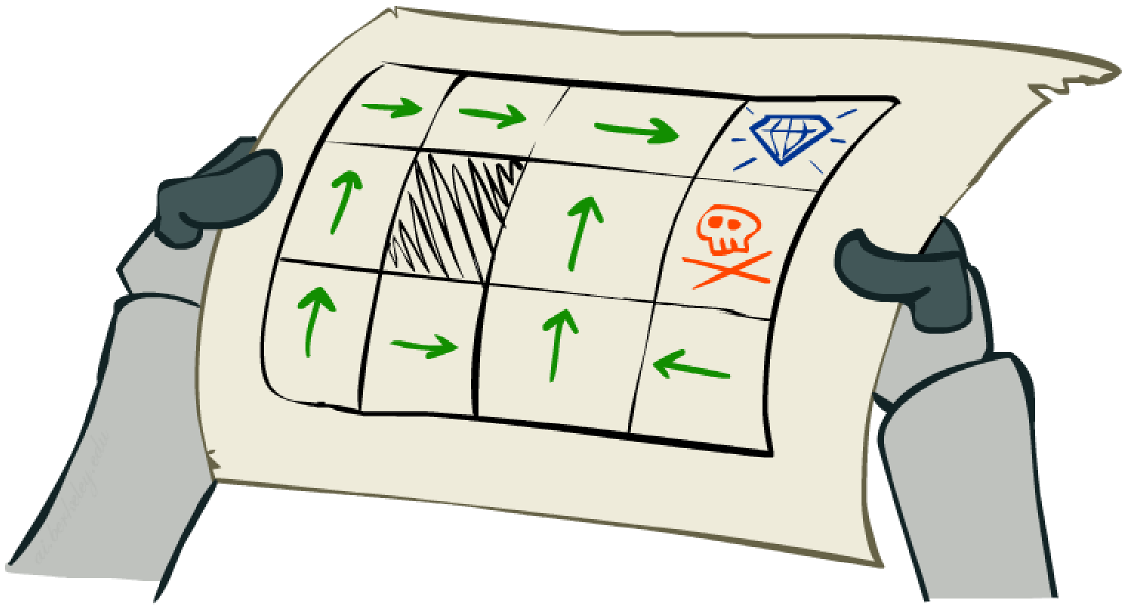
$$[s_0, s_1, . . . , s_n], [s_0, s_2, . . . , s_m], ..., [s_0, s_4, . . . , s_n]$$

  - So, to evaluate π, we need to measure the expected utility of the possible environment histories generated by that policy

Optimal policy π^*

- Optimal policy $\pi^*: S \rightarrow A$
  - An optimal policy is a policy that yields the highest expected utility

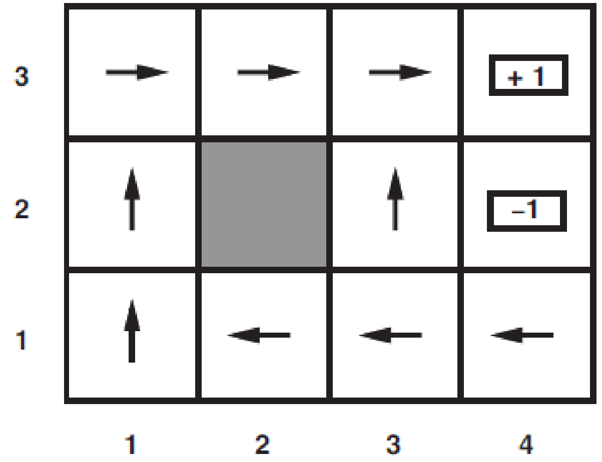




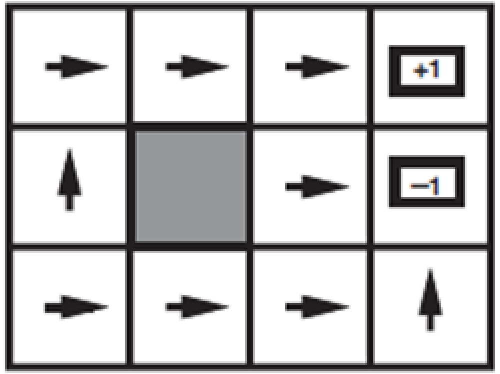
Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

Optimal policies: balancing risk and reward

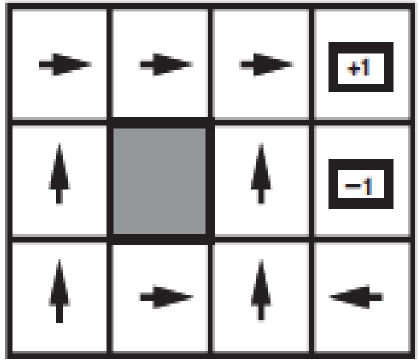
- Optimal policy changes with choice of living rewards R(s).



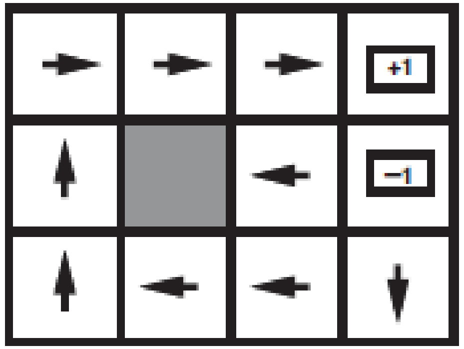
$R(s) = -0.04$



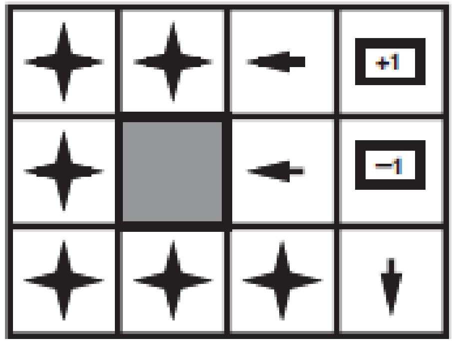
$R(s) < -1.6284$



$-0.4278 < R(s) < -0.0850$



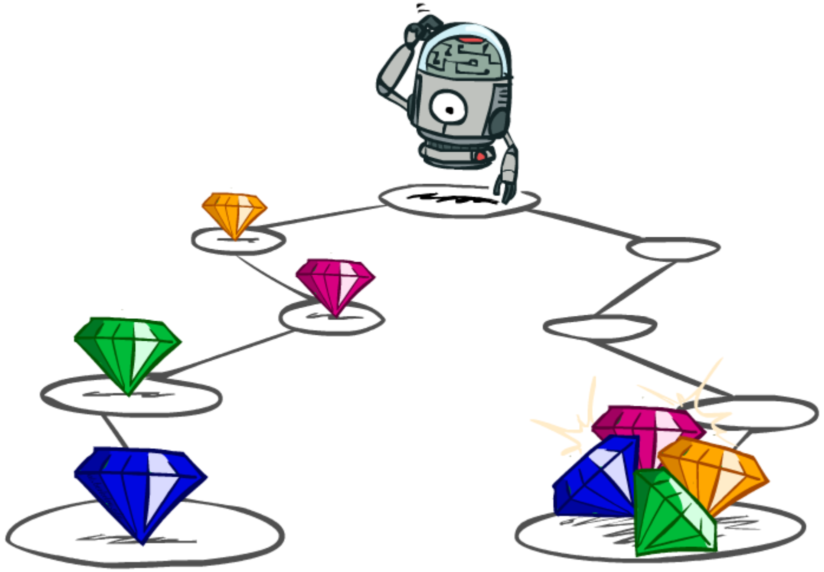
$-0.0221 < R(s) < 0$



$R(s) > 0$

Utility of the agent

- Utility of the agent: depends on the sequence of states, i.e., environment history $[s_0, s_1, \ldots, s_n]$, rather than a on a single state
  - Utility function: $U_h([s_0, s_1, \ldots, s_n])$
- How to calculate utilities for state sequences (in order to be able to select the best)?
  - More or less? [1, 2, 2] or [2, 3, 4]
  - Now or later? [0, 0, 1] or [1, 0, 0]



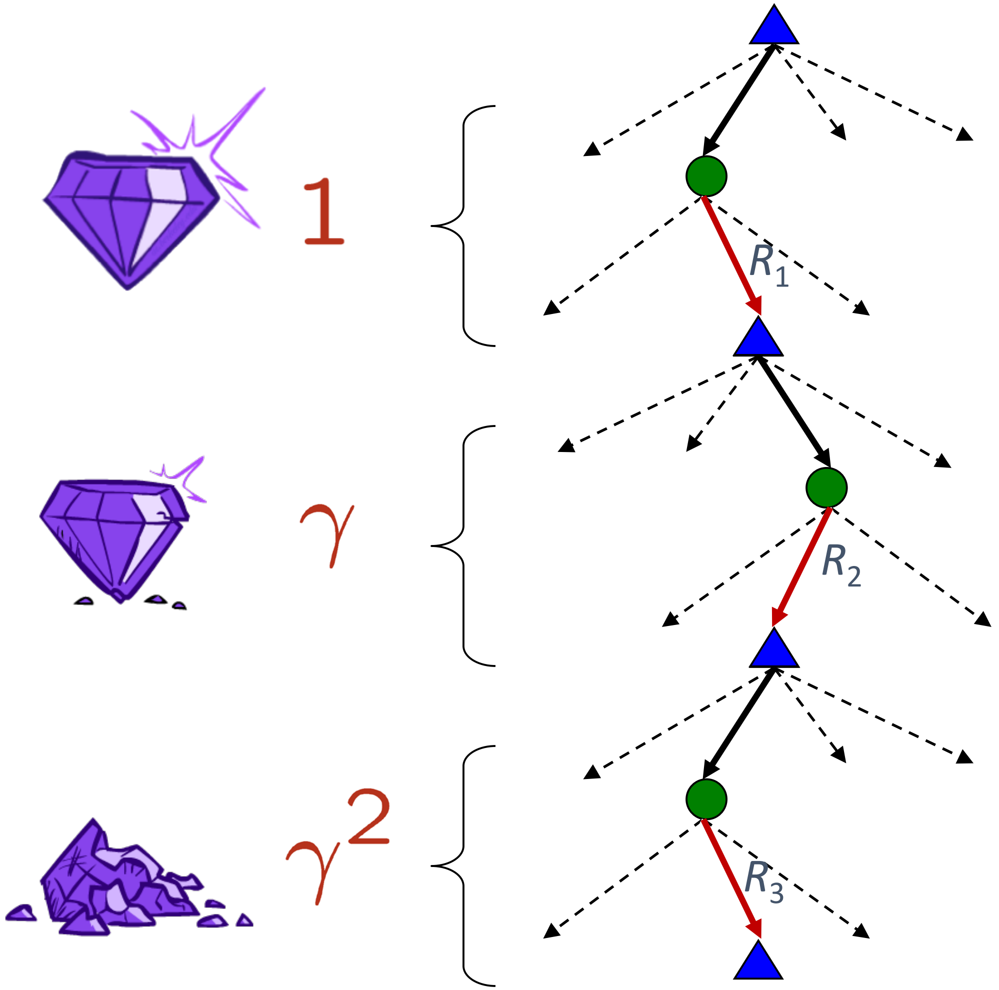
- It is reasonable to maximize the sum of rewards
- It is also reasonable to prefer rewards now to rewards later
- Idea → discount factor $\gamma \in [0, 1]$

Discounting

- Utility is a sum of discounted rewards:

  $$U[s_o, s_1, \ldots] = R(s_o) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots + \gamma^n R(s_n)$$

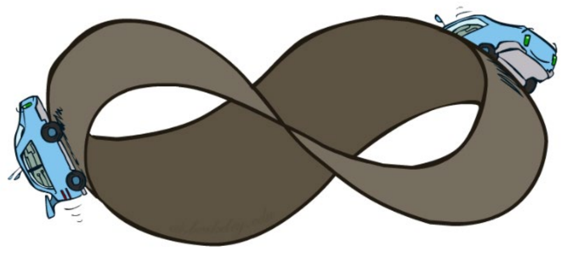
- The order of rewards matters
  - Example: discount of 0.5
    - U([1,2,3]) = 1 + 0.5*2 + 0.25*3
    - U([3,2,1]) = 3 + 0.5*2 + 0.25*1
    - U([1,2,3]) < U([3,2,1])
- Smaller γ means shorter-term focus
  - Example: discount of 0.1
    - U([1,2,3]) = 1 + 0.1*2 + 0.01*3



Why discounting

- Sooner rewards probably do have higher utility than later rewards
- Helps with the infinite sequences problem
  - What if the game lasts forever (infinite sequence)? Do we get infinite utilities?
  - It is also hard to compare state sequences with infinite utilities
  - It can be shown that if $\gamma < 1$ and rewards are bounded by $+/-R_{max}$

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



- Also helps our algorithms to converge

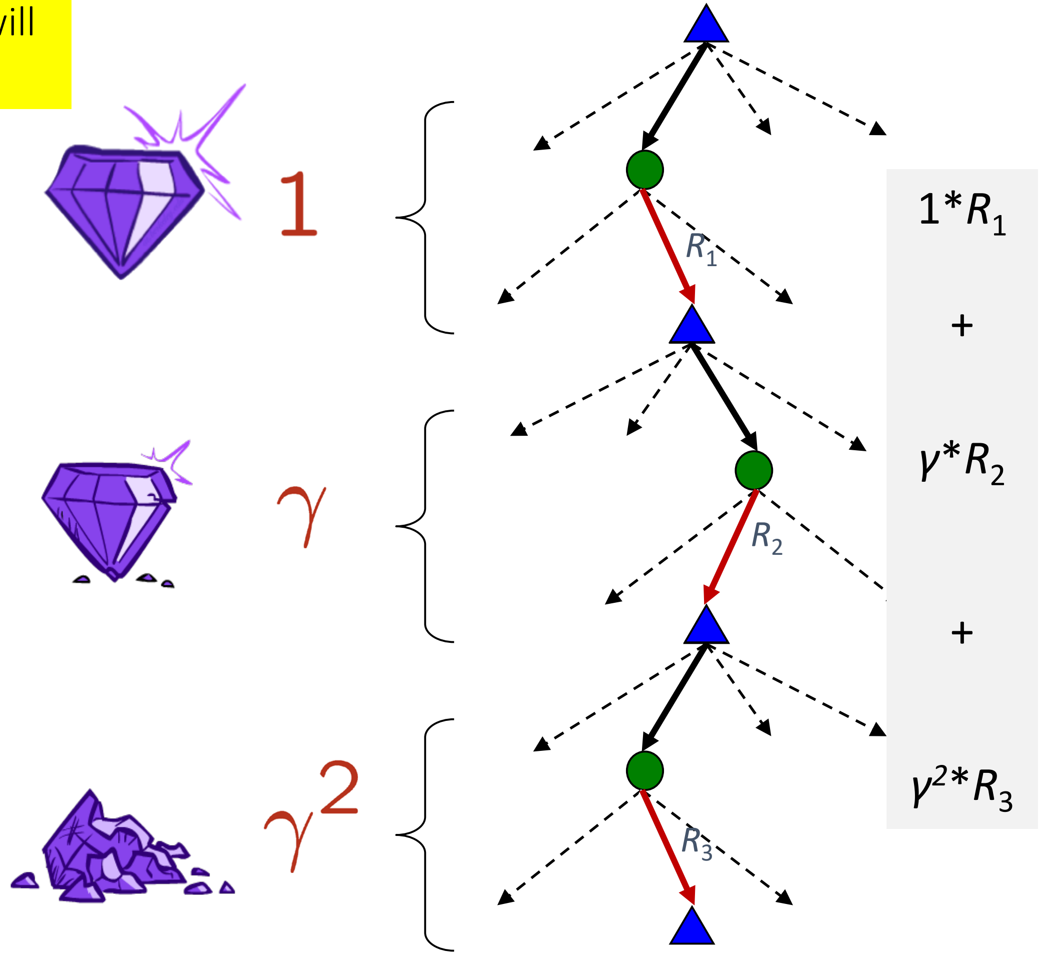
Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

Solving MDPs

For the moment, we assume a full model of the world - We will relax this definition later

- Input: the MDP formulation
  - 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)
- Output
  - An optimal policy: $\pi^*: S \rightarrow A$ that maps each state to an action, $\pi^*(s)$
  - If followed by the agent π^* will yield the maximum expected total reward or utility
- Utility = sum of (discounted rewards)

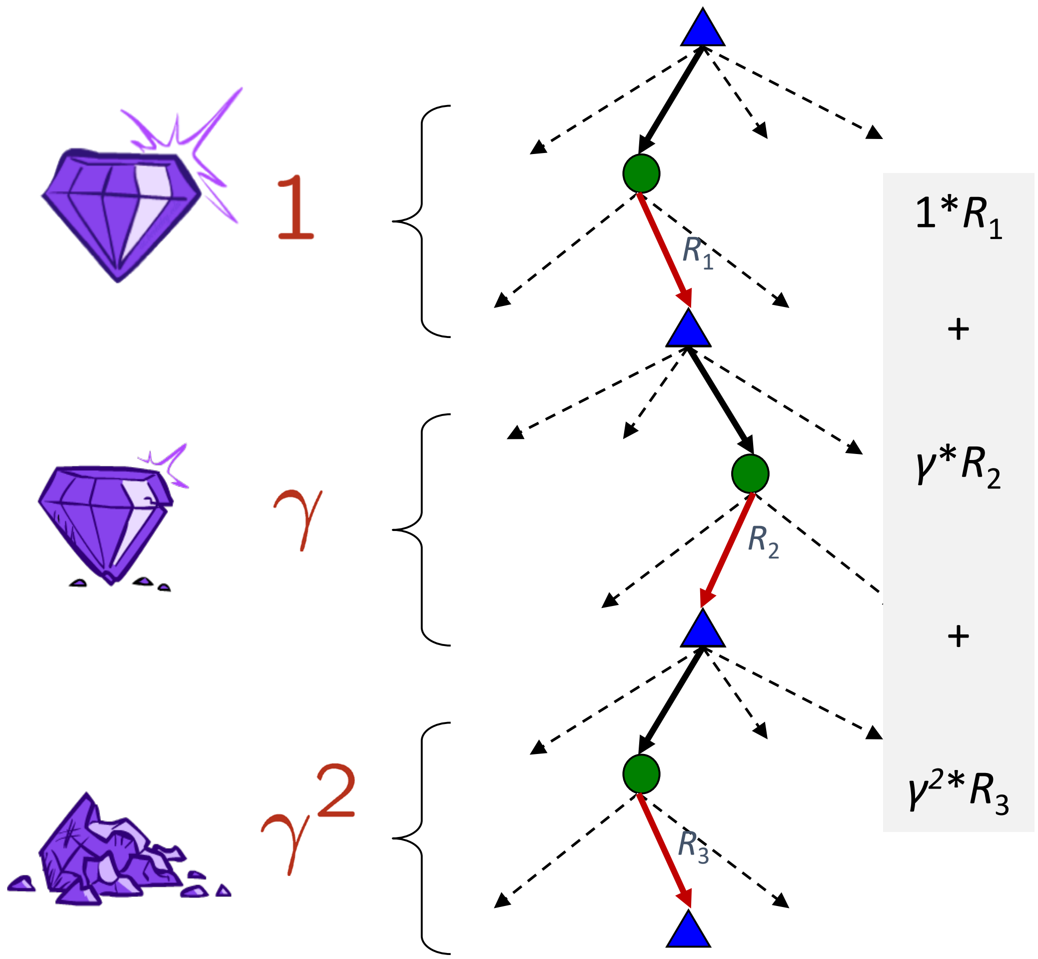


Value function (A major component of an RL agent)

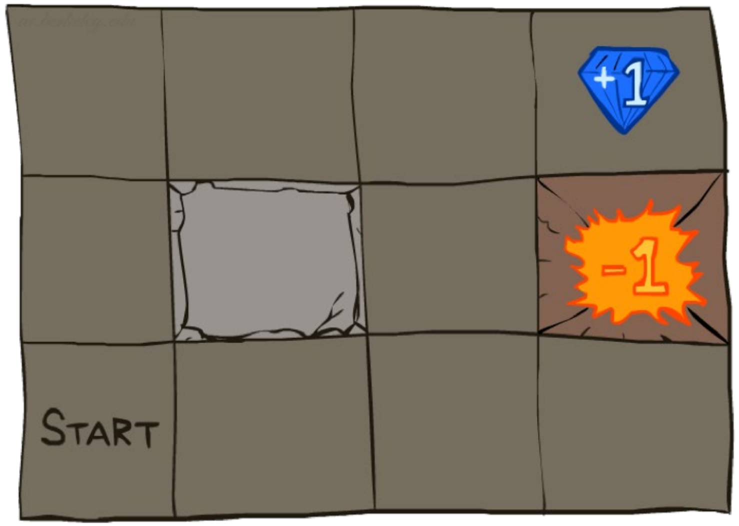
- How we decide among possible actions/states?
- The value(utility) of a state *s* → *V(s)* (called V-value)
  - It is a prediction of future reward
  - Used to evaluate the goodness/badness of states

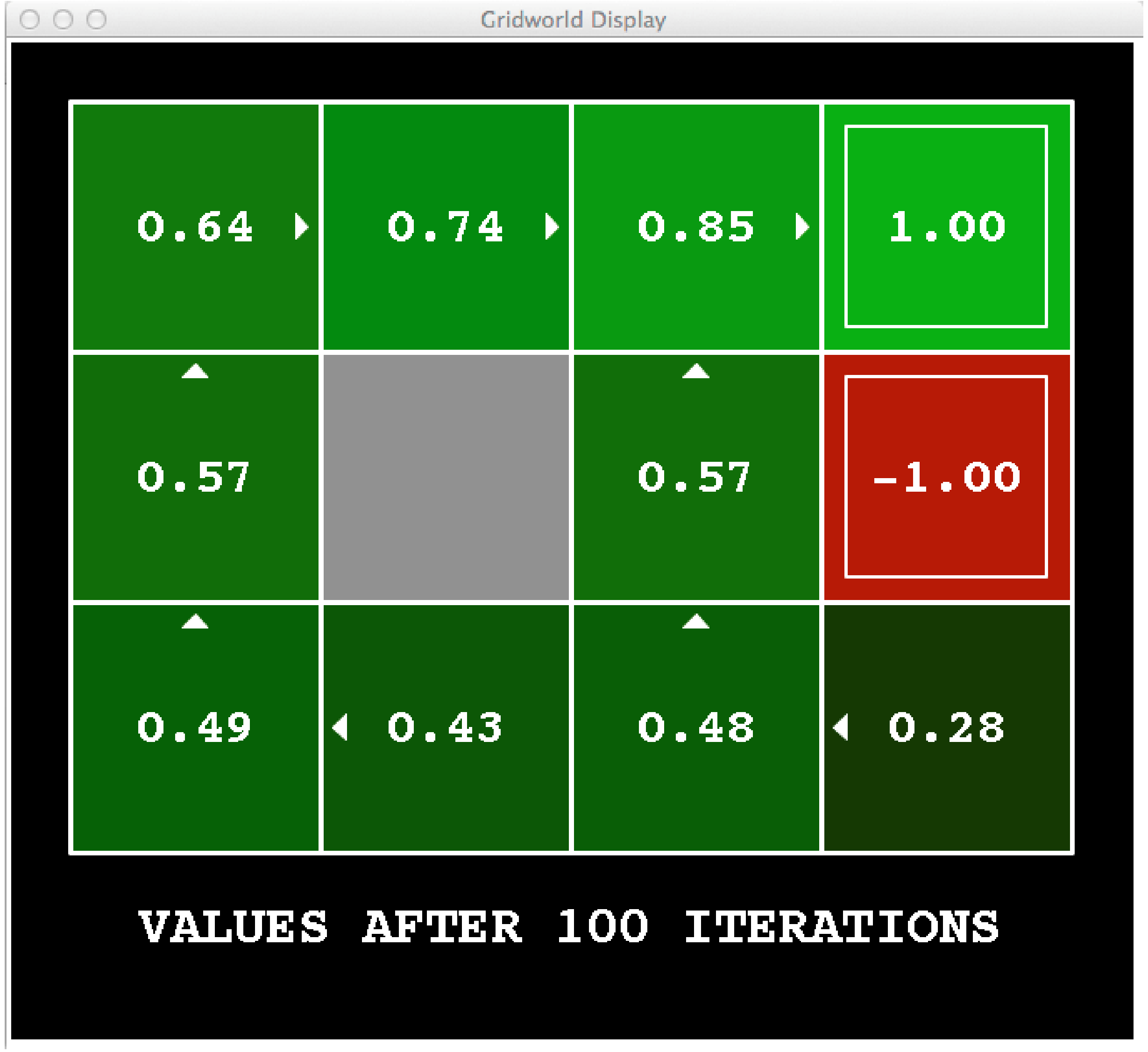
- It is the expected value of the state

$$v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right]$$



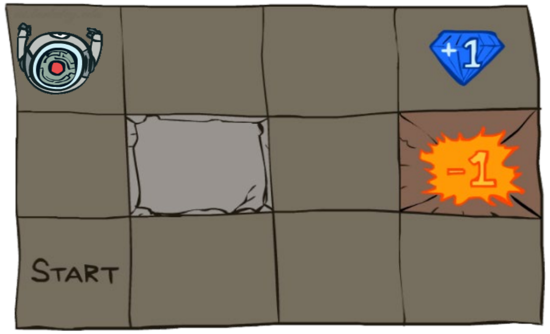
Gridworld example: V values - utilities of states s

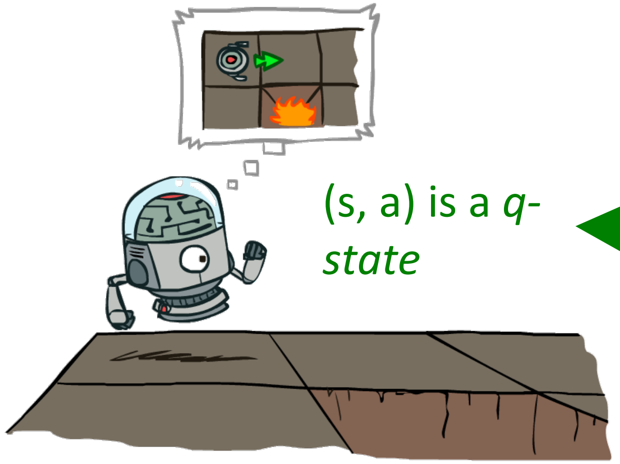


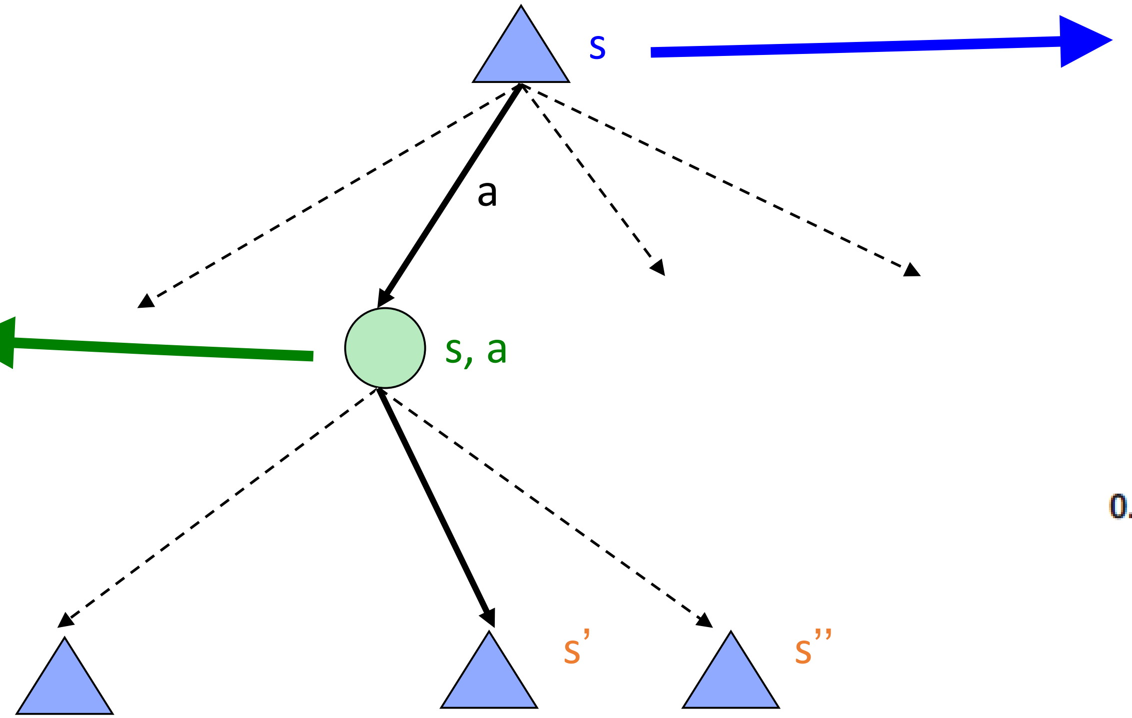


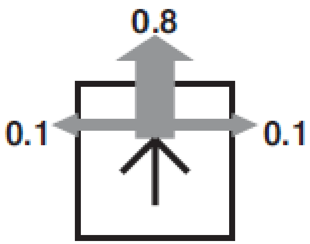
Q-states

- The combination of a state *s* and action *a,* denoted by *(s,a)*, is called a Q-state
  - It represents being in state *s* and having taken action *a*
  - still, due to uncertainty, we don't know what the outcome of the action will be
- The value of a q-state is called Q-value

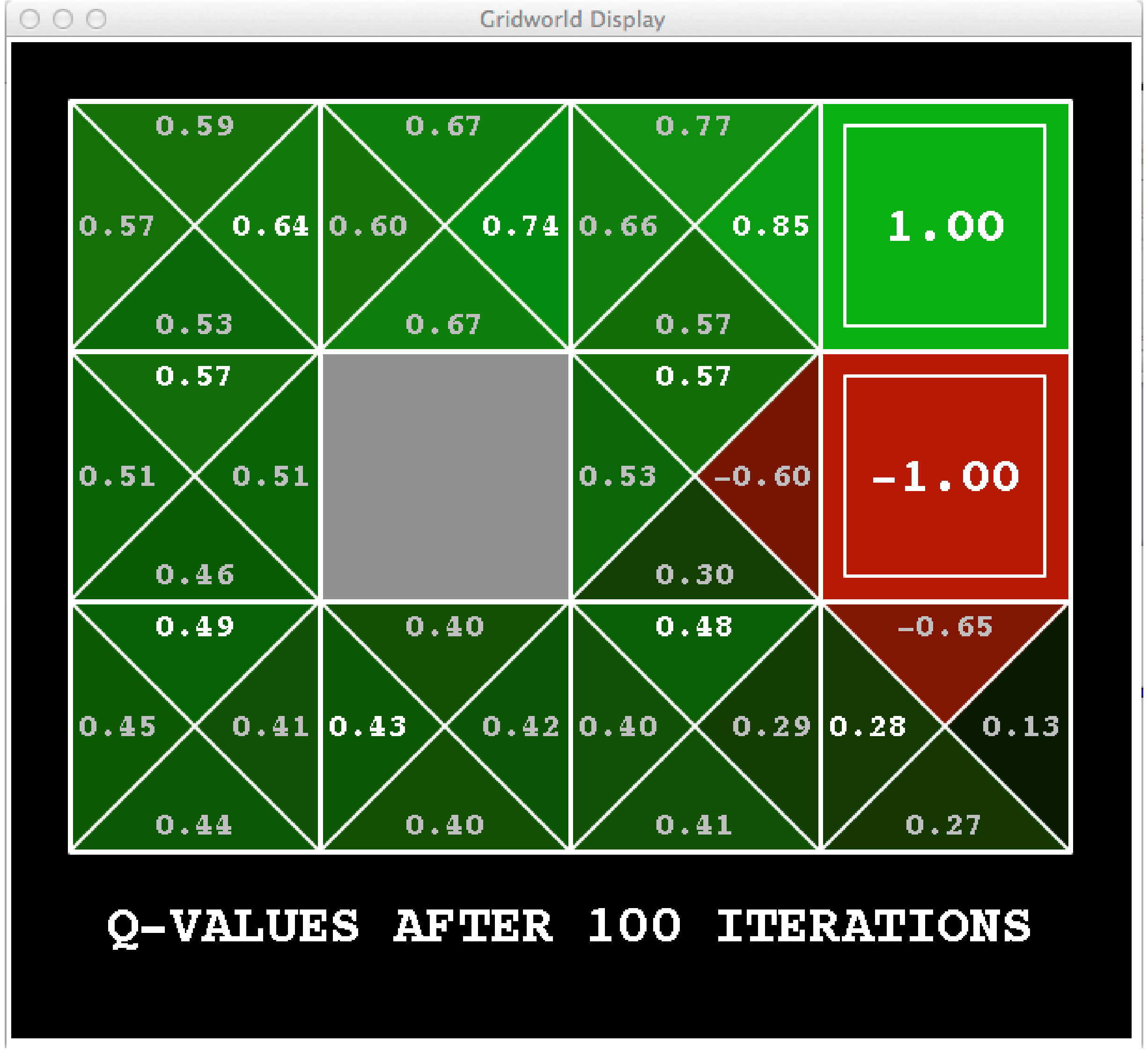




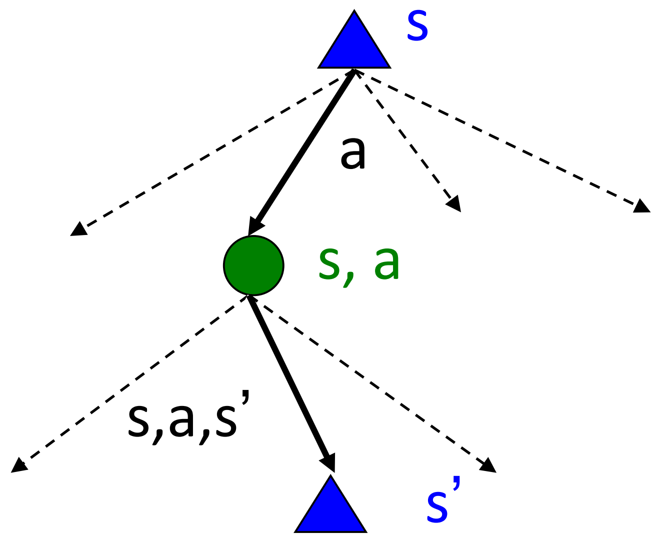




Gridworld example: Q values - utilities of states (s,a)



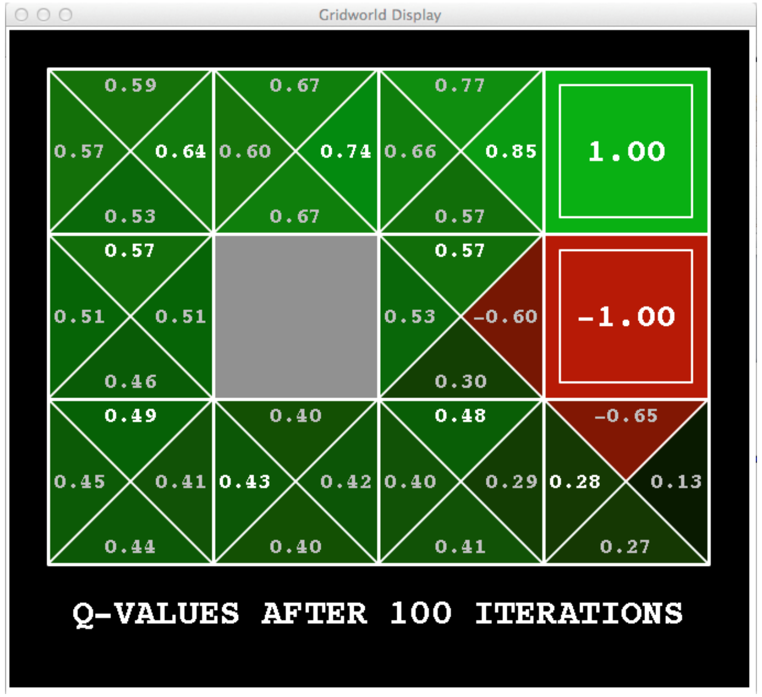
Optimal V* and Q* quantities



- The value (utility) of a state *s*:
  - $V^*(s)$ = expected utility starting in *s* and acting optimally from that point onwards
- The value (utility) of a q-state *(s,a)*:
  - $Q^*(s,a)$ = expected utility starting out having taken action *a* from state *s* and acting optimally from that point onwards

- Acting optimally: a rational agent should choose the action that maximizes the expected utility of the subsequent state
  - $\pi^*(s)$ = optimal action from state *s*

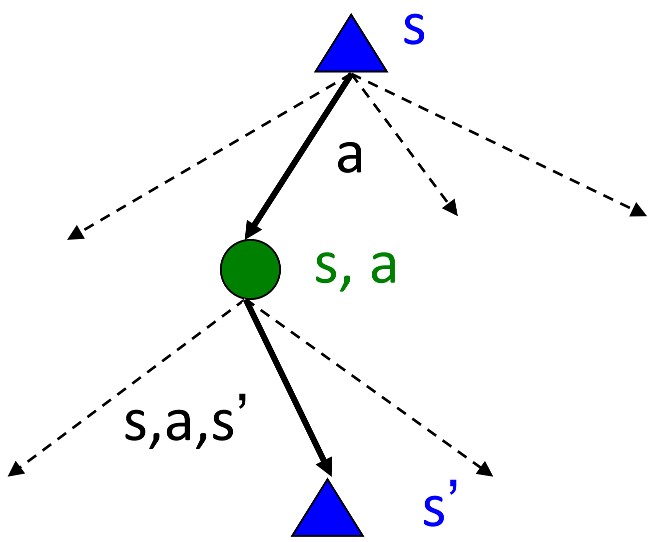
$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a)$$



Bellman equations

- The utility of a state is the immediate reward *R(s,a,s')* for that state plus the expected discounted utility of the next state *γV*(s')*, assuming that the agent chooses the optimal action.

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



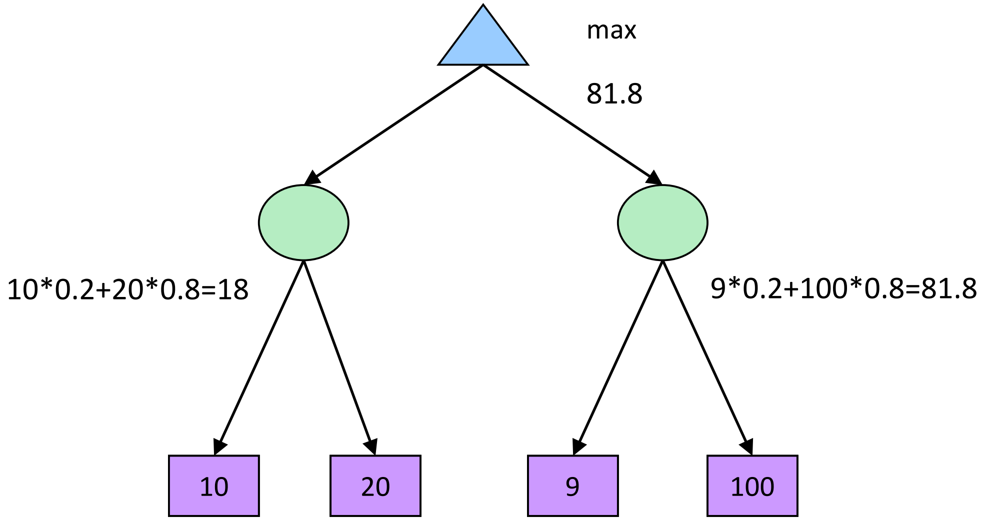
  - Due to the non-deterministic actions, we should compute the expected value
  - We assume the agent acts optimally

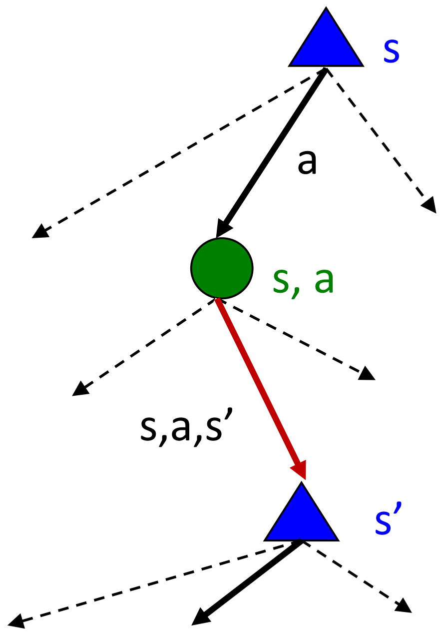
$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

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

V- and Q- values

- Small example assuming γ=1, R=0





s is a *state*

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

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

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

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

The value Iteration algorithm

- The Bellman equation is the basis of the value iteration algorithm for solving MDPs
- If there are n possible states, then there are n Bellman equations, one for each state.

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

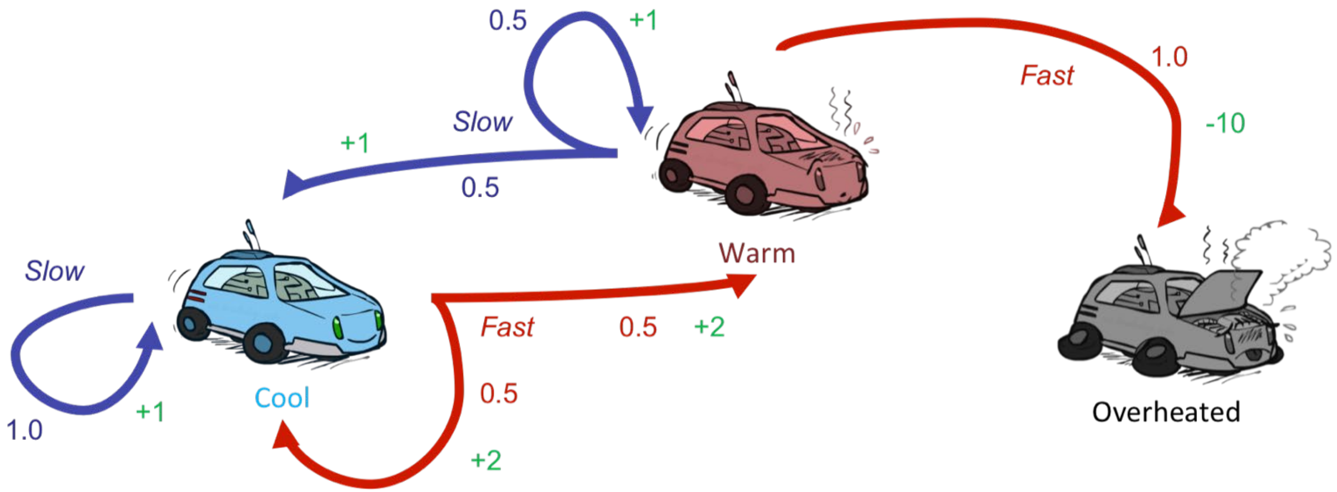
- We want to solve them simultaneously, but they are non-linear (max operation)
- Try an iterative approach
  - start with arbitrary initial values for the utilities
  - At each iteration k+1, for all states s
    - Update $V_{k+1}(s)$ based on $V_k(s)$ → Bellman update

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

  - Repeat this until convergence

Example: Value Iteration

V_2	3.5	2.5	0
V_1	2	1	0
	0	0	0



V_0

Assume no discount (γ=1)!

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

V_1(cool)=max{1*[1+0],0.5*[2+0]+0.5*[2+0]}=2

V_2(cool)=max{1*[1+2],0.5*[2+2]+0.5*[2+1]}=max{3,3,5}=3,5

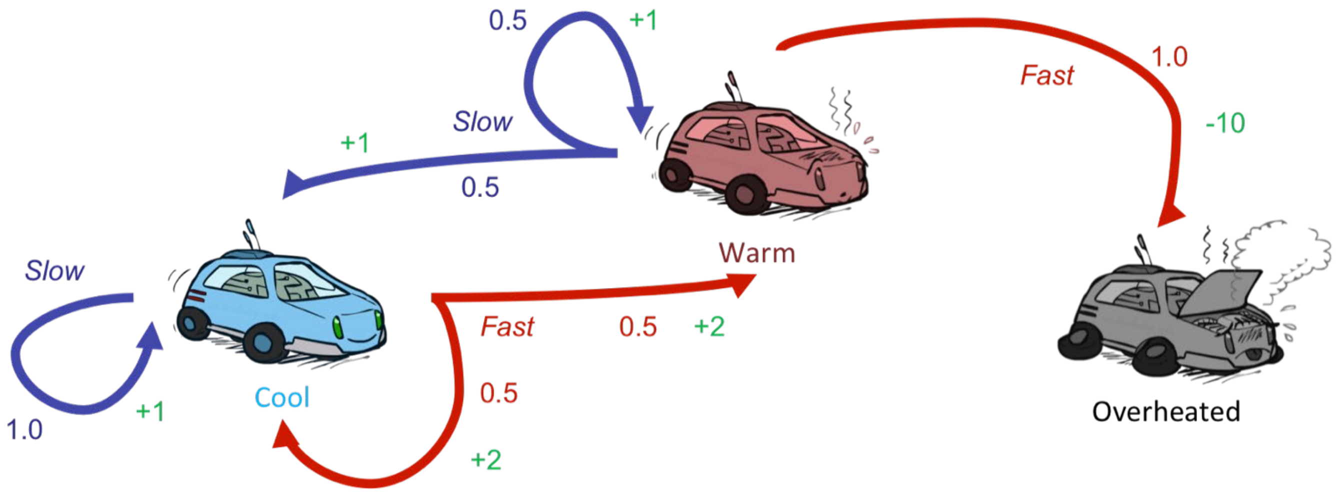
Example: Value Iteration

V_2

V_1

0 0 0

V_0

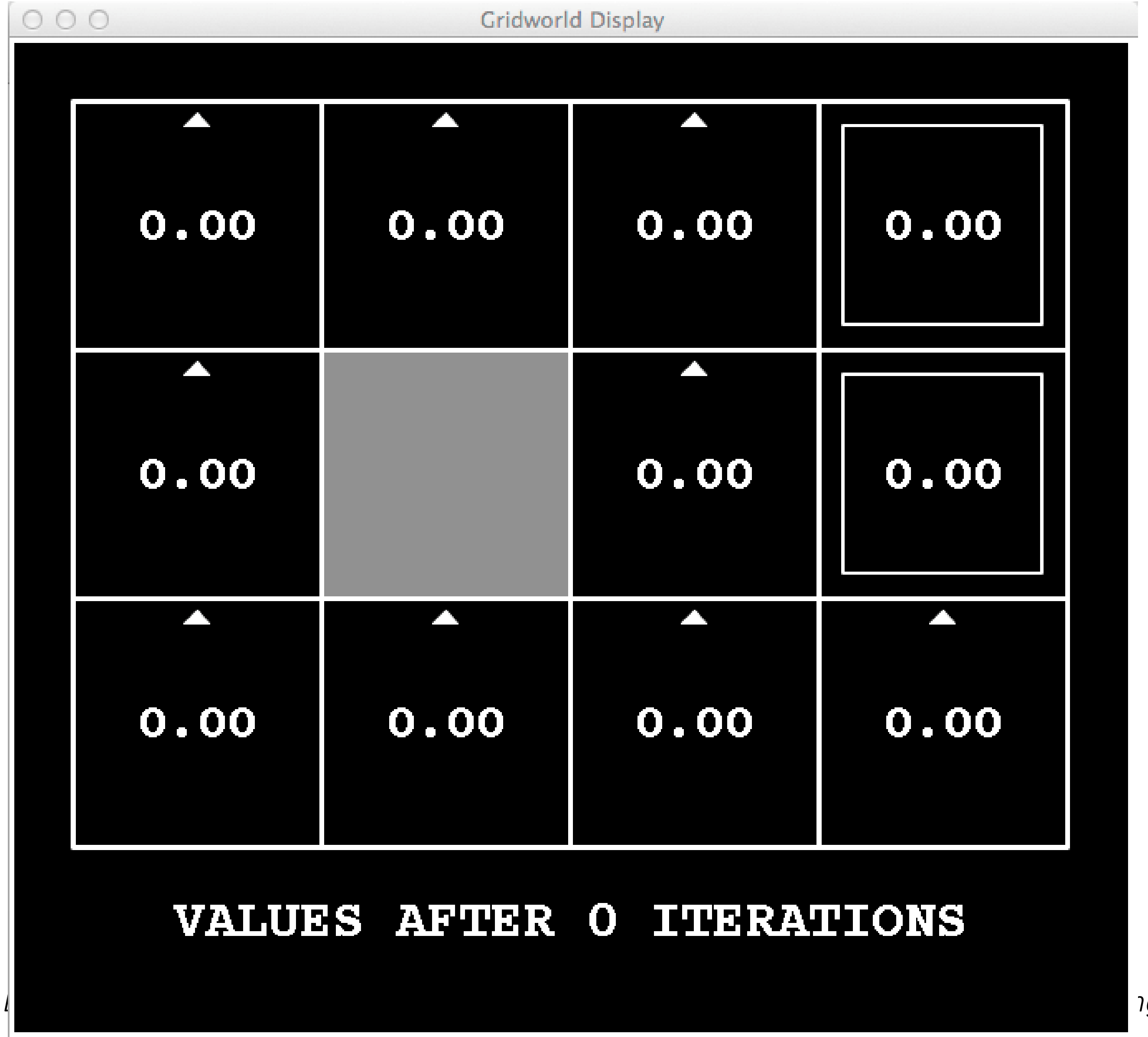


Assume a discount γ=0.5!

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

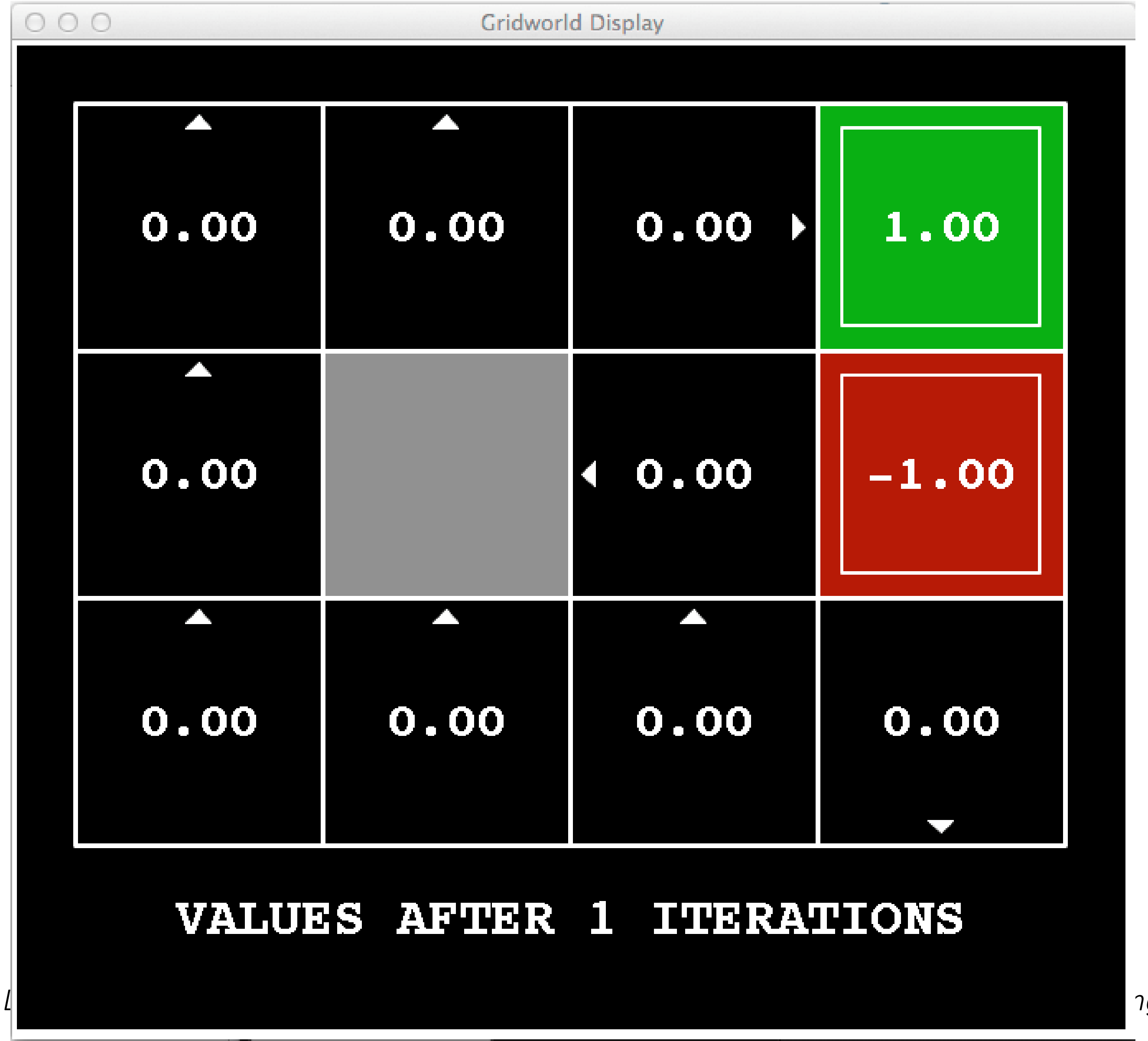
Homework!

Example: k=0



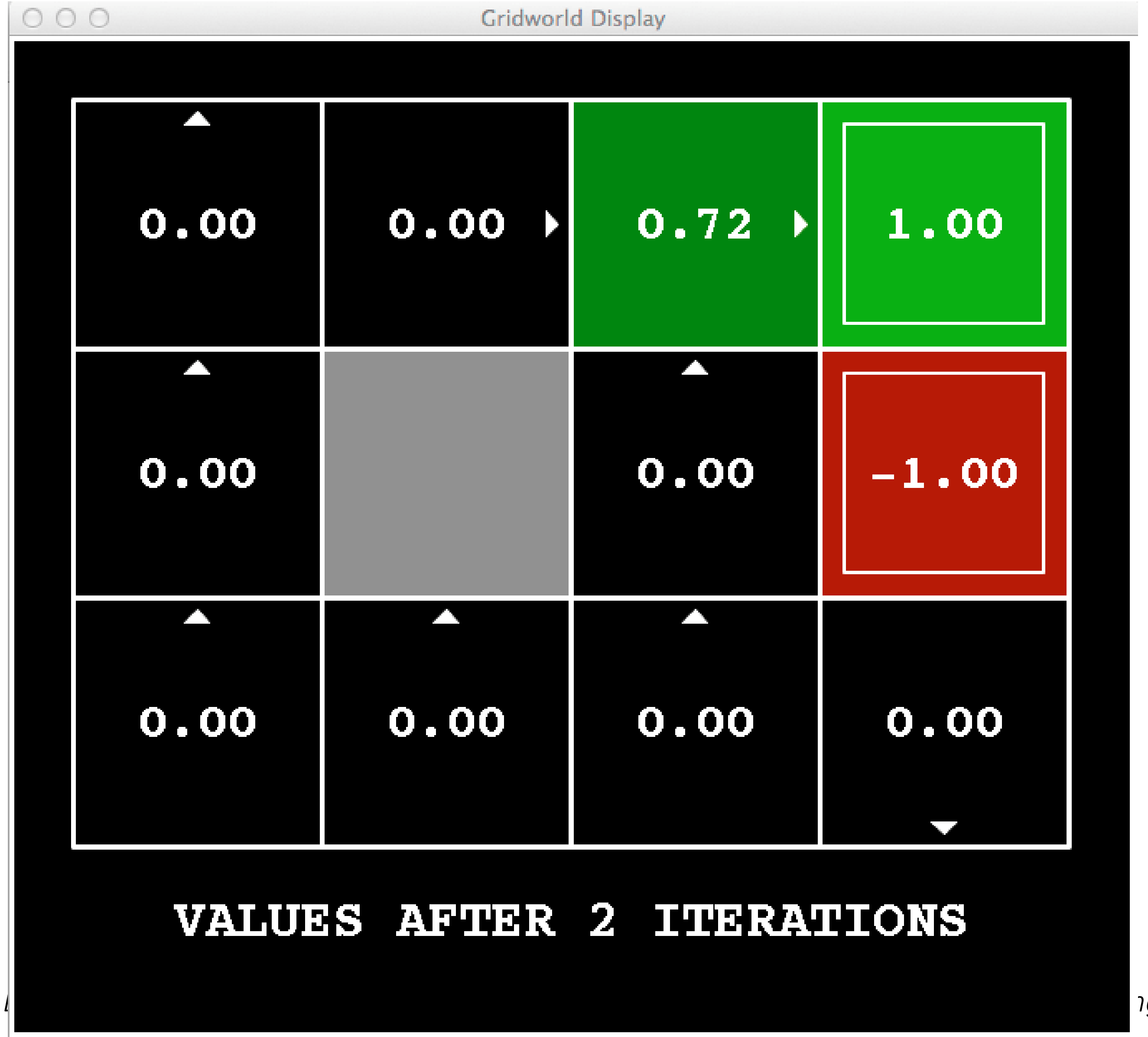
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=1



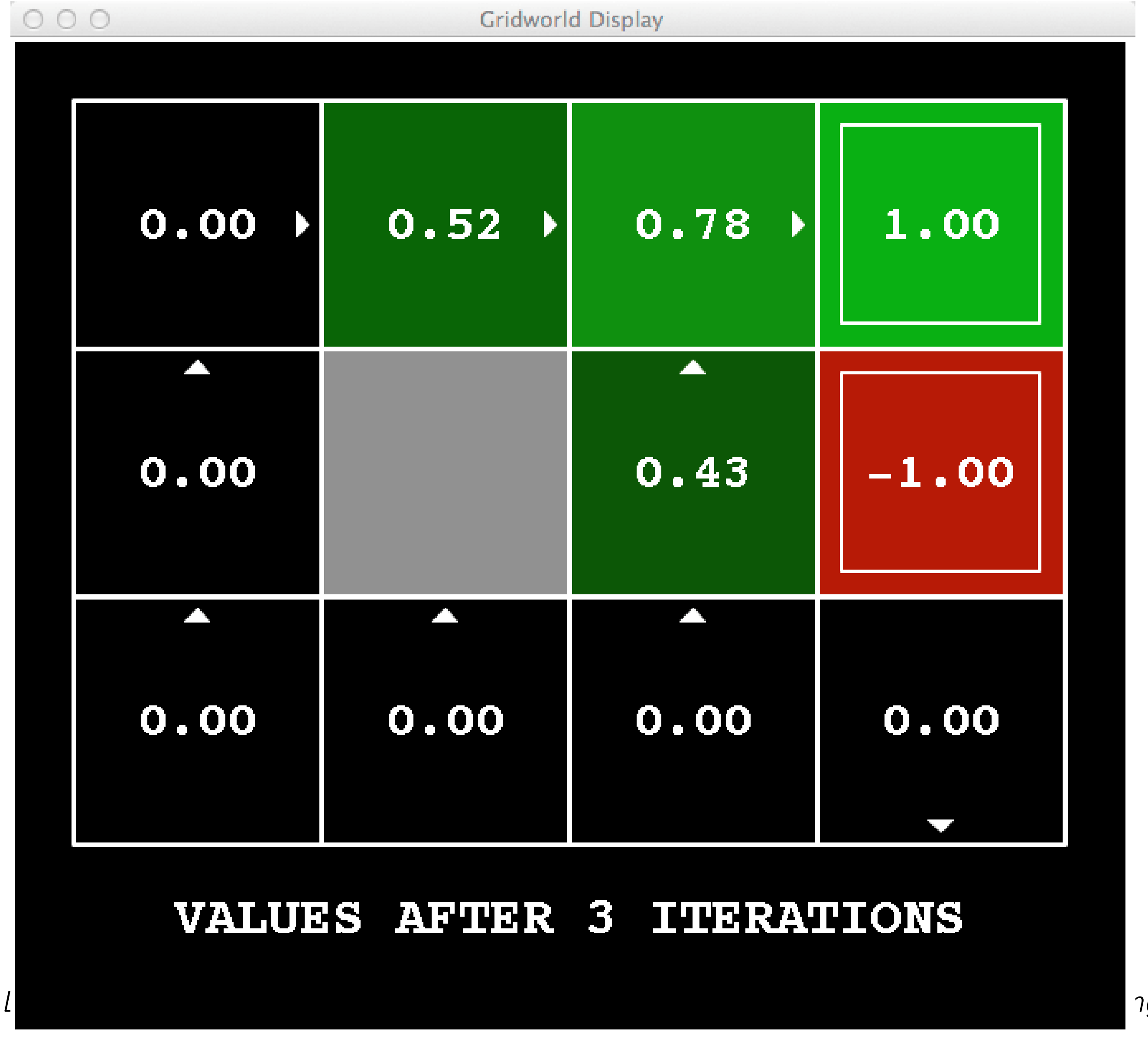
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=2



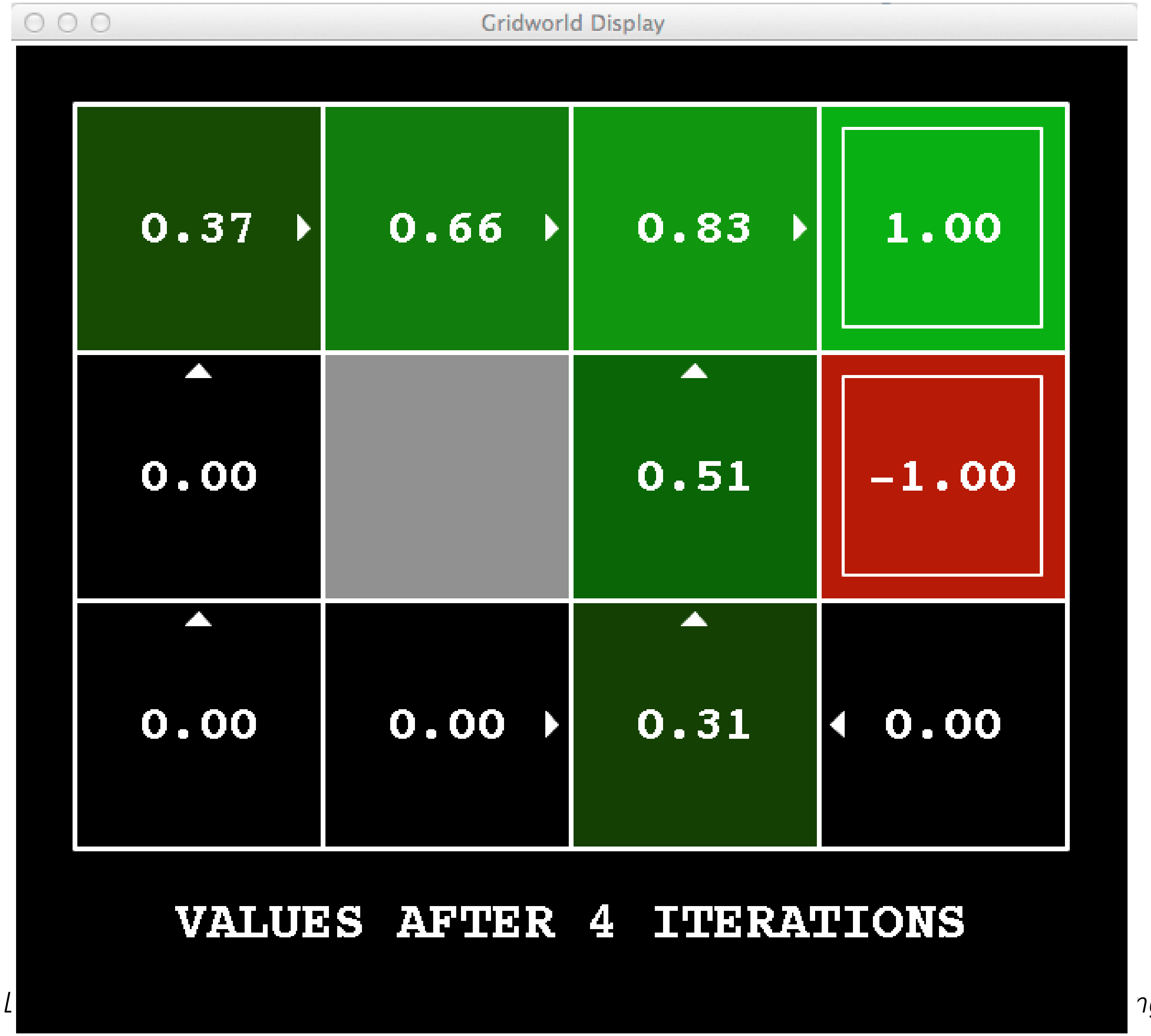
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=3



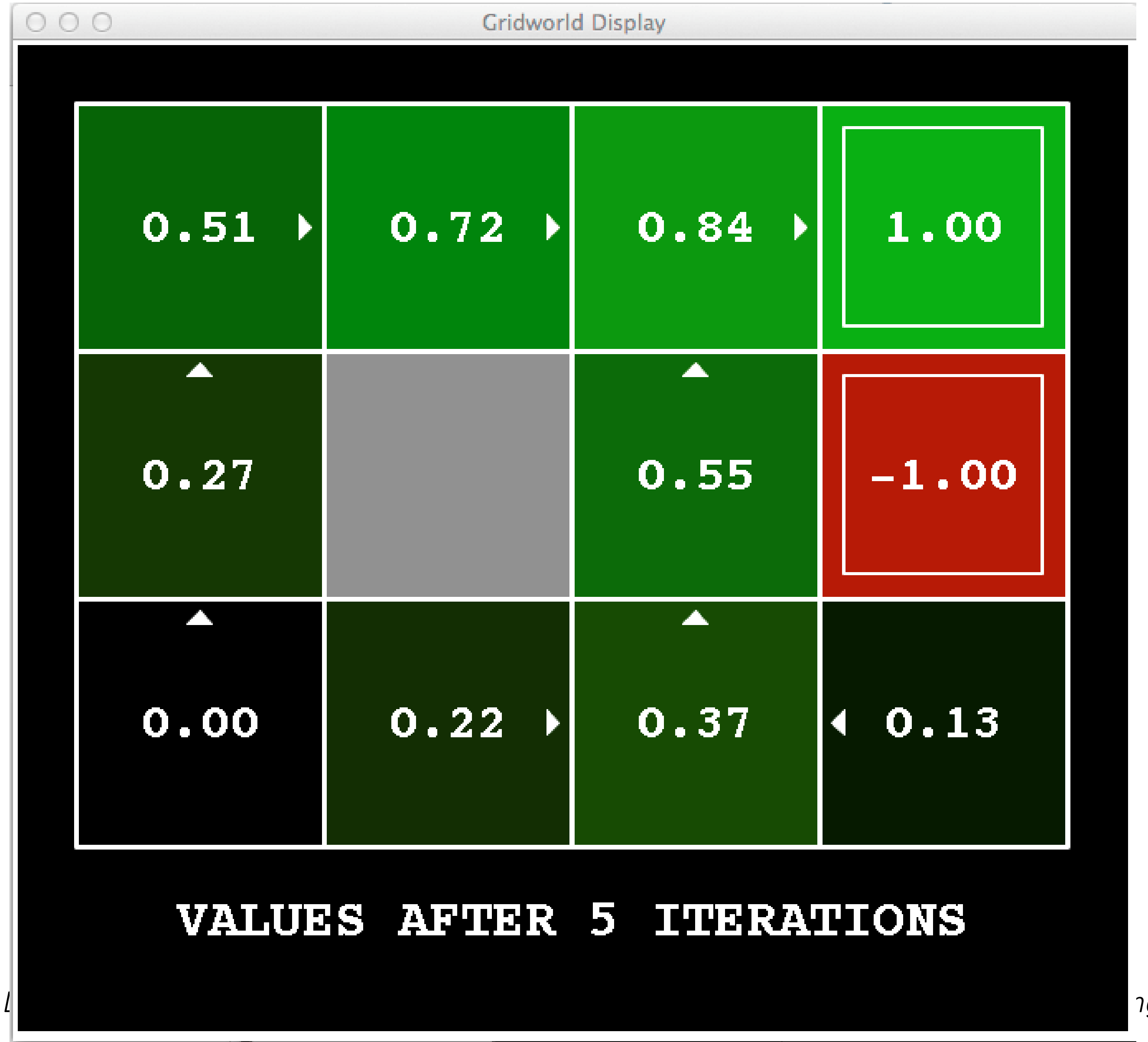
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=4



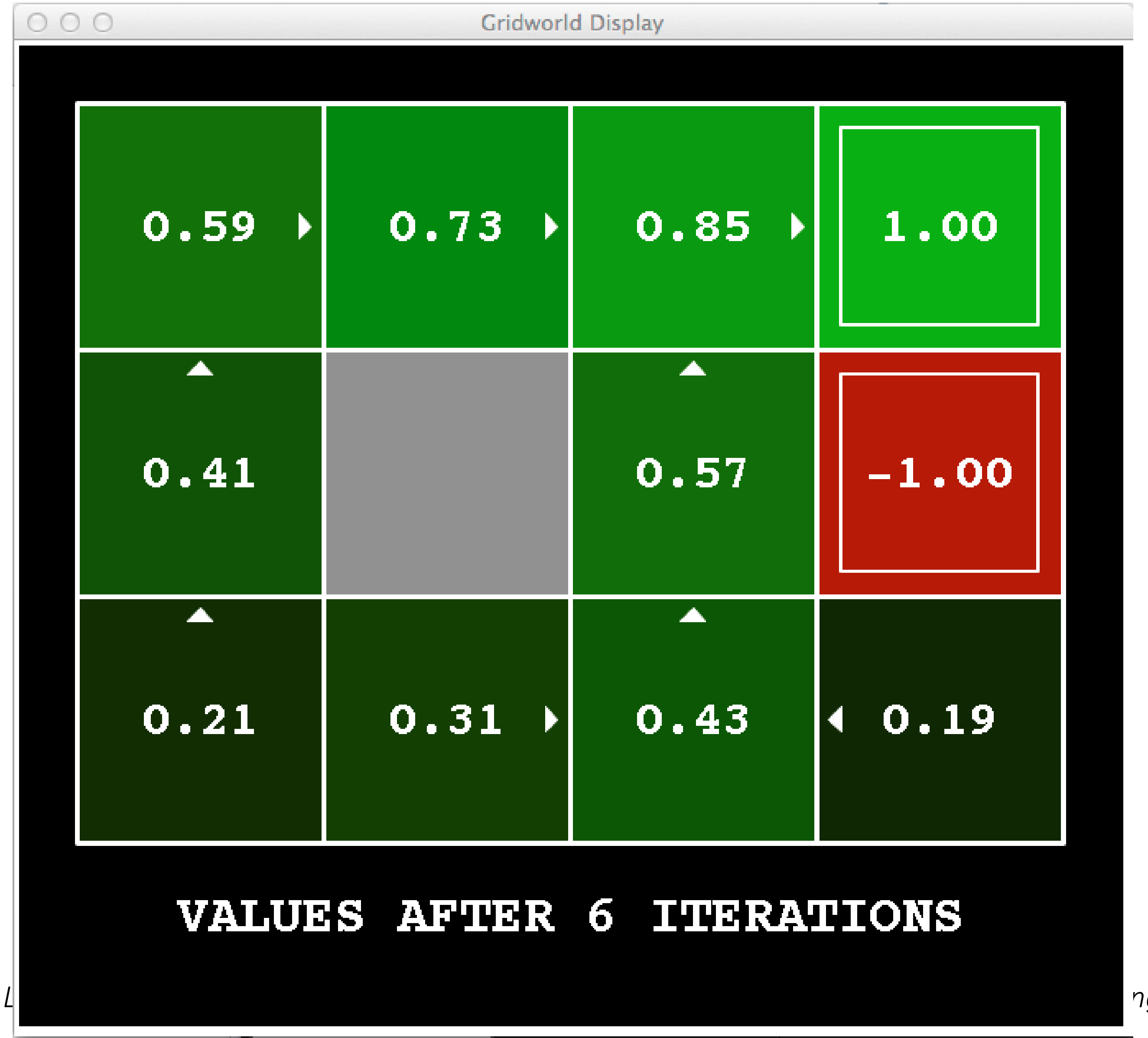
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=5



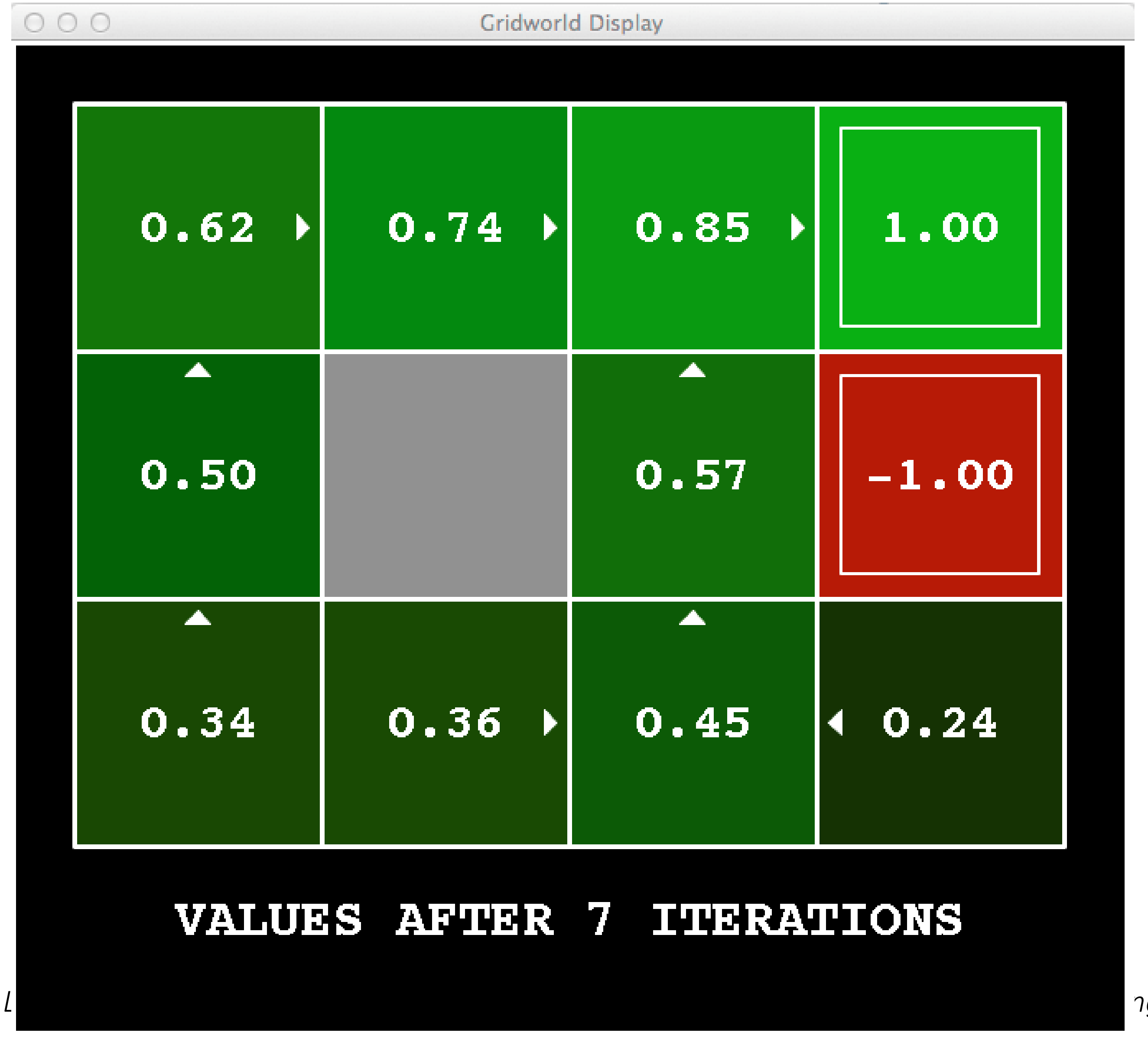
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=6



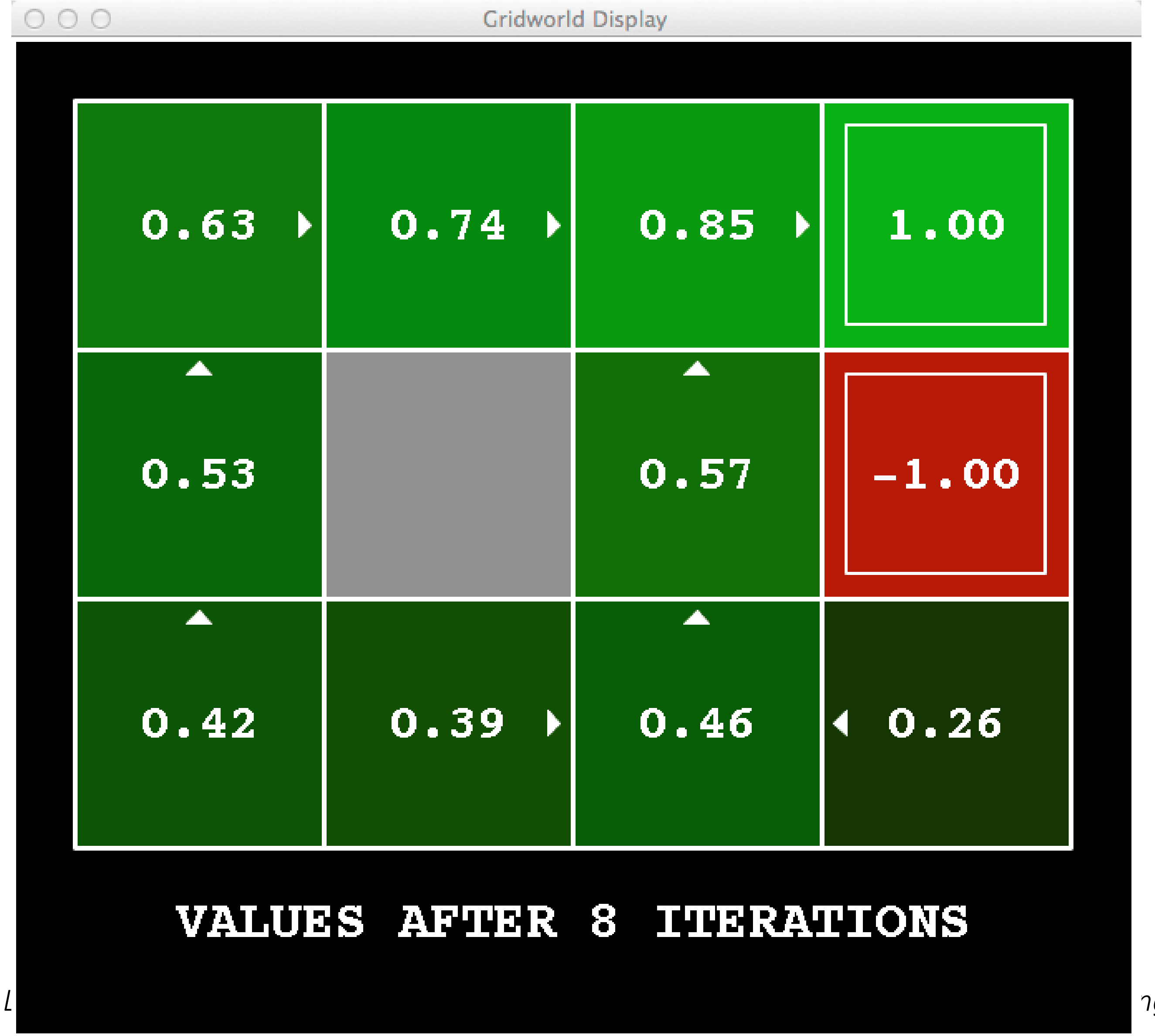
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=7



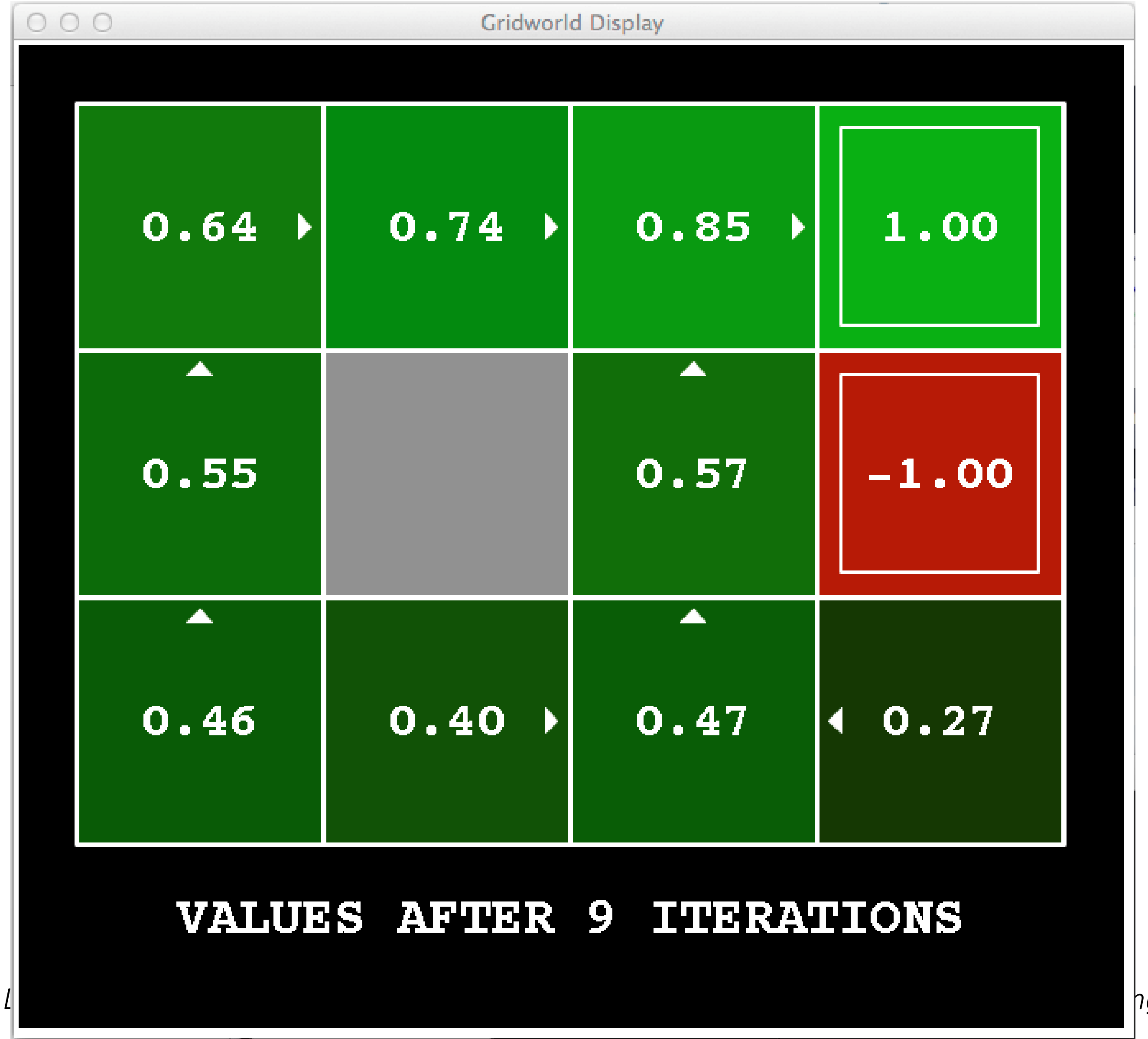
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=9

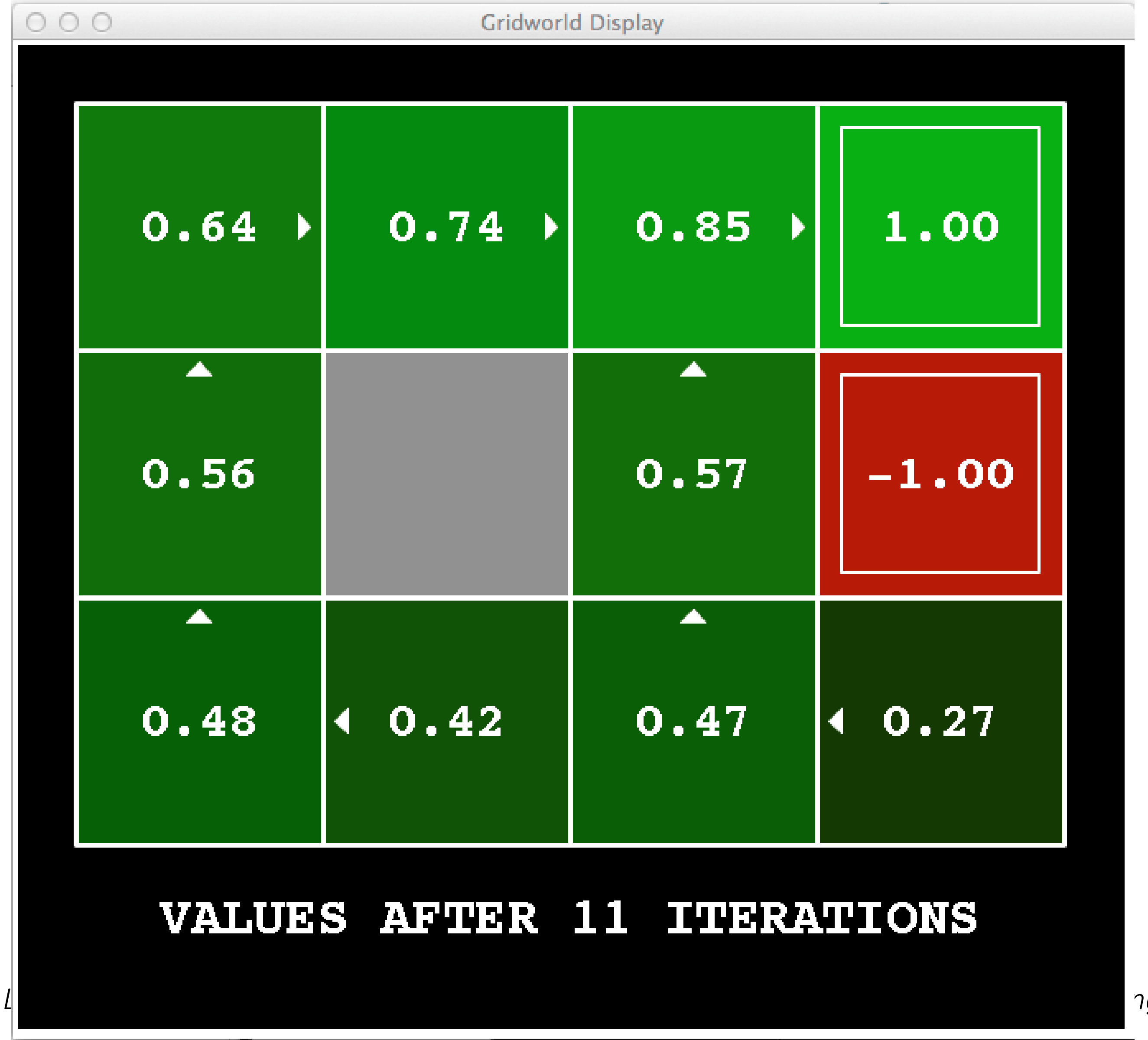


Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=10

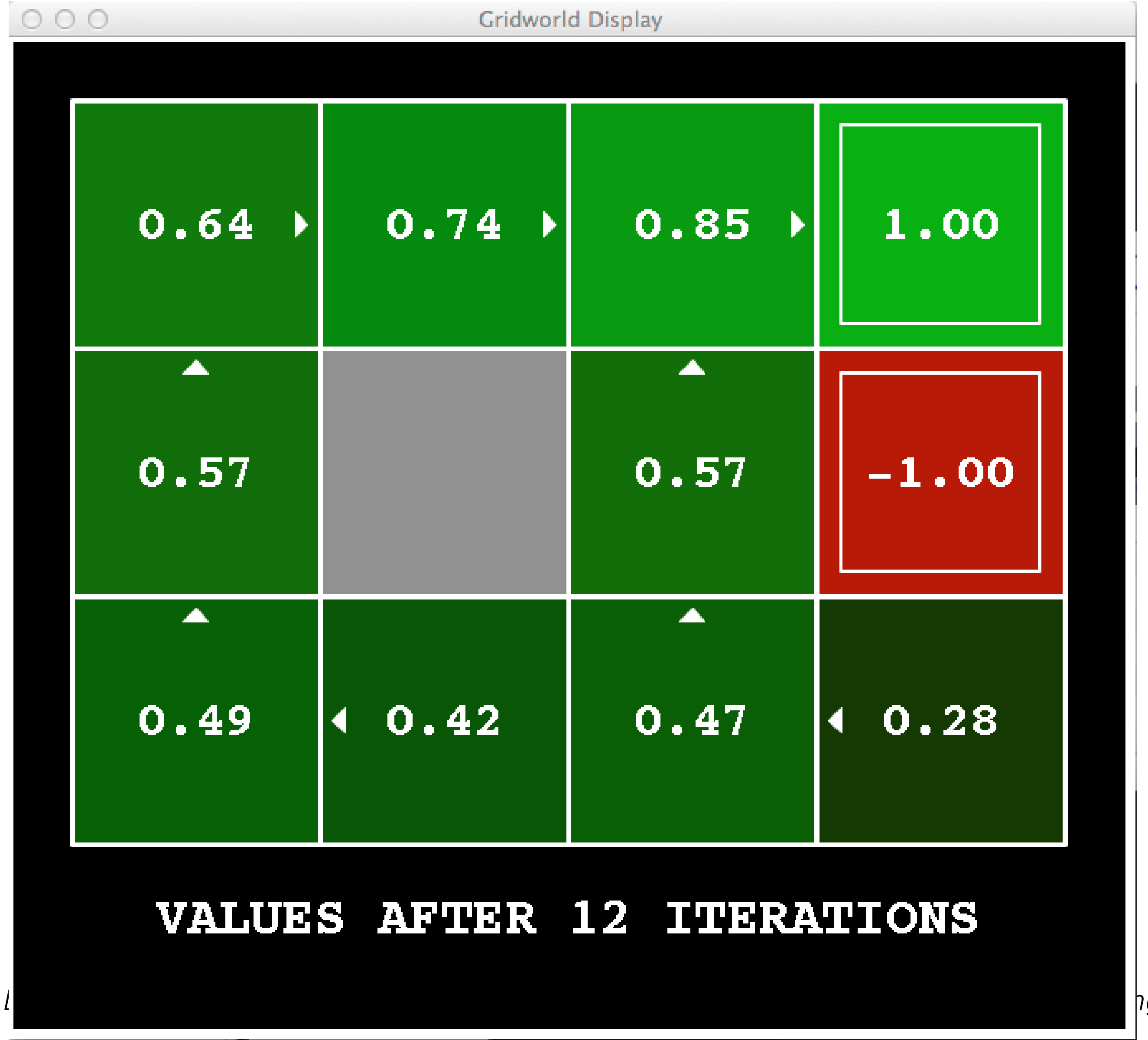
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=11



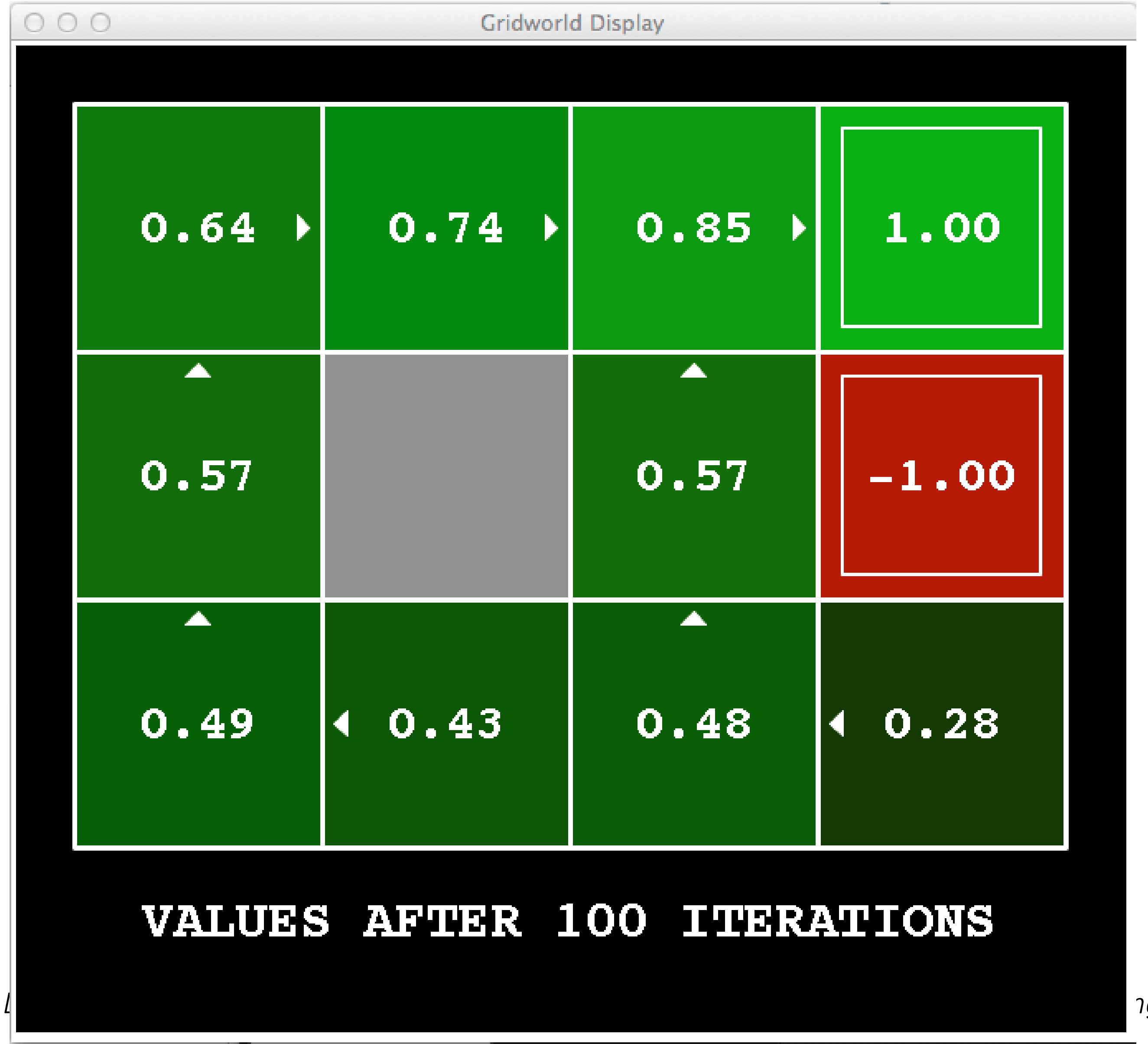
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=12



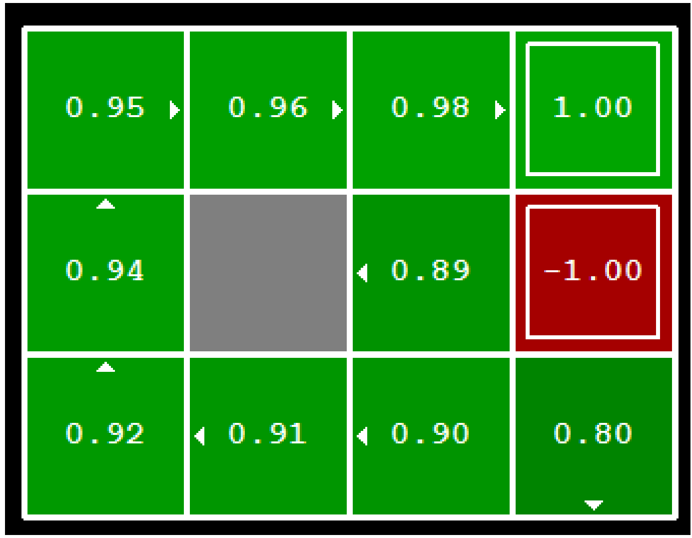
Noise = 0.2
Discount = 0.9
Living reward = 0

Example: k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

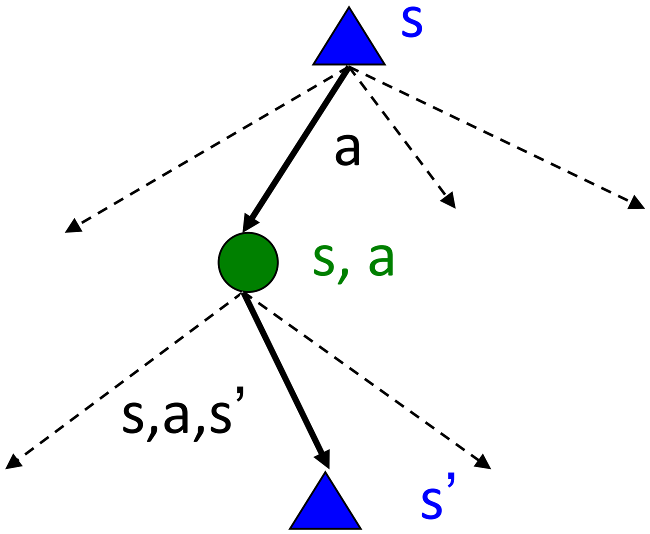
From optimal V* values to optimal policy π*()



- At convergence we find $V^*(s)$ based on which we can find policy $\pi^*(s)$
- Policy extraction: given the optimal values, what is the implied optimal policy?
- The V-values are non-actionable, we need to look 1-step ahead

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

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

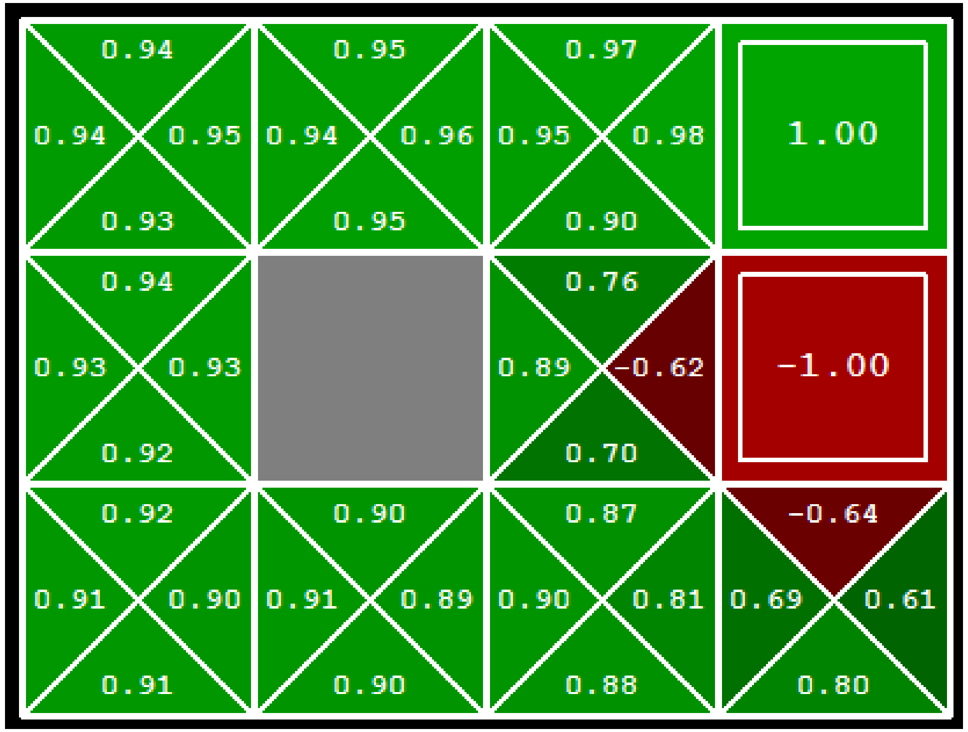


Policy extraction from Q*-values

- If we have *q*-values, completely trivial to decide
  - Select the action that takes us to the *q*-state with the max *q*-value

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

- So, it is better to keep the *q*-values



Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

From value-iteration to policy-iteration

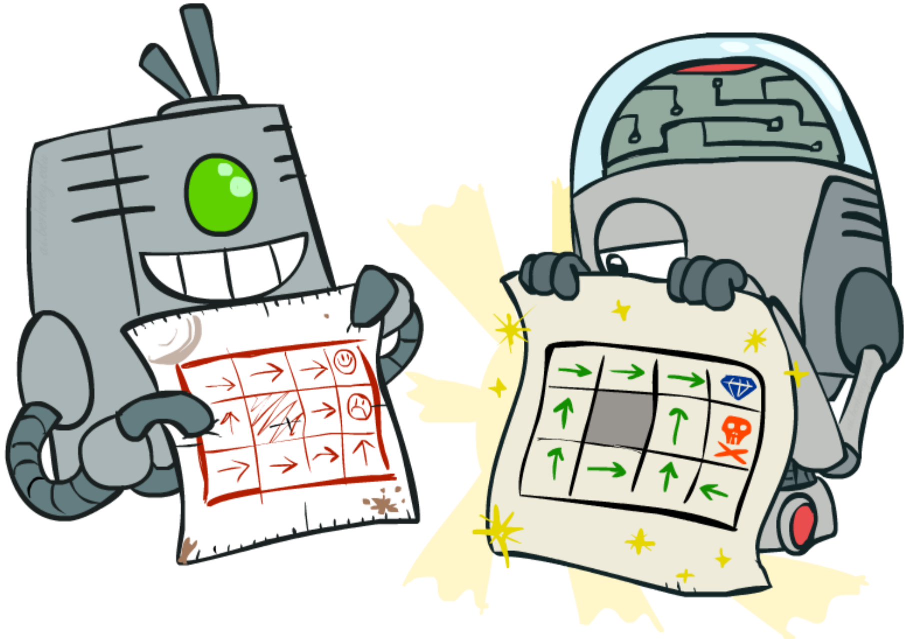
- In value iteration, approximations get refined towards optimal values
- But value convergence takes too long
- Policy might converge faster than values

- So, it is possible to get an optimal policy even if the utility function estimates are inaccurate

Policy iteration

- The policy iteration algorithm consists of two steps
  - Step 1. Policy evaluation: calculate utilities for some fixed policy $\pi()$ (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 1, 2 until convergence



Step 1: Policy evaluation

- Given a fixed policy π calculate the utility of each state s if π were to be executed
  - $V^{\pi}(s)$: the utility of s according to π

Do the optimal action $\pi^*()$ | Do what π says to do

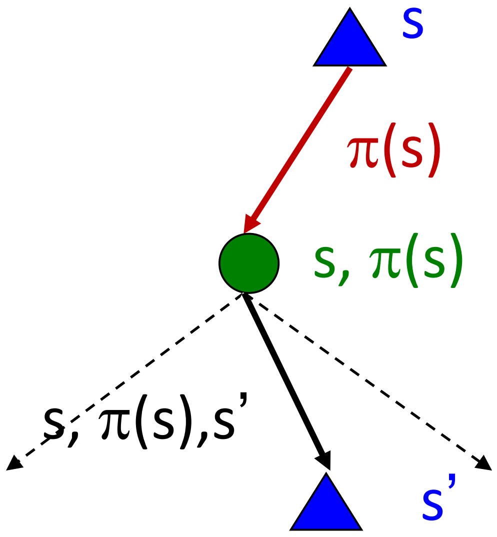
- For the optimal action we need to take max over all actions to compute the optimal values
- If we have a fixed policy $\pi()$, we only need <u>one</u> action per state $\pi(s)$
  - of course, the result depends on which policy we fixed

Step 1: Policy evaluation

- 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 / simplified Bellman equation):

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



compare it to optimal policy approach

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

Step 1: Policy evaluation - computation

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

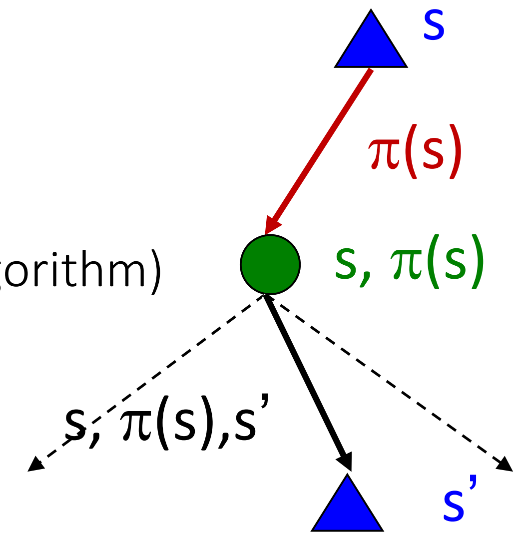
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

- Idea 1: Turn recursive Bellman equations into updates (like value iteration algorithm)

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

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

  - Much faster than the value iteration approach

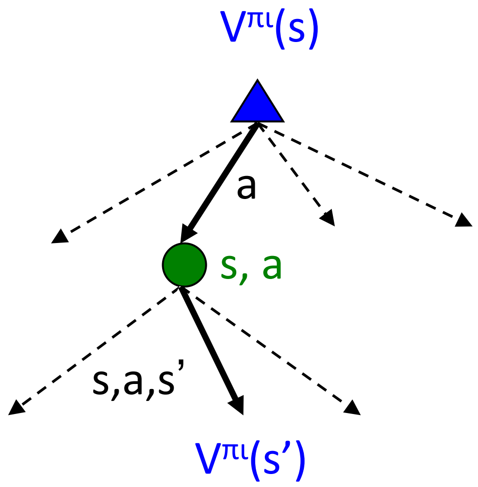


- Idea 2: Without the max operation, the Bellman equations are just a linear system.
  - We can solve them using exact solution methods (if state space is small)

Step 2: Policy Improvement

- We can evaluate a fixed policy π (using policy evaluation) → $V^{\pi}(s)$
- How can we improve π?
- Policy improvement: with fixed values $V_{\pi}(s)$, find the best action according to one-step-look-head (so, use 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 Iteration algorithm

- Step 1 – Policy evaluation: with fixed current policy π, find values with policy evaluation
  - Iterate until values converge (simplified Bellman update formula):

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

  - Note: could also solve value equations with other techniques
- Step 2 – Policy improvement: with fixed values, find the best action according to one-step-look-head (so, use policy extraction)

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

- Repeat steps 1, 2 until convergence
- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster (under some conditions)

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

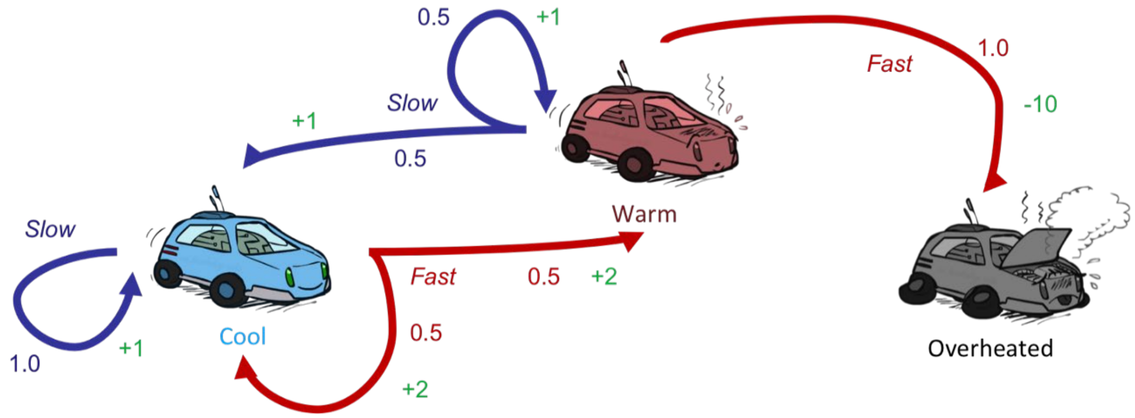
- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They differ only in whether we plug in a fixed policy or max over actions

Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

From MDPs to Reinforcement Learning

- In a MDP, we have
  - A set of states $s \in S$
  - A set of actions (per state) A
  - A transition model T(s,a,s')
  - A reward function R(s,a,s')
- and we are looking for a policy $\pi(s)$



From MDPs to Reinforcement Learning

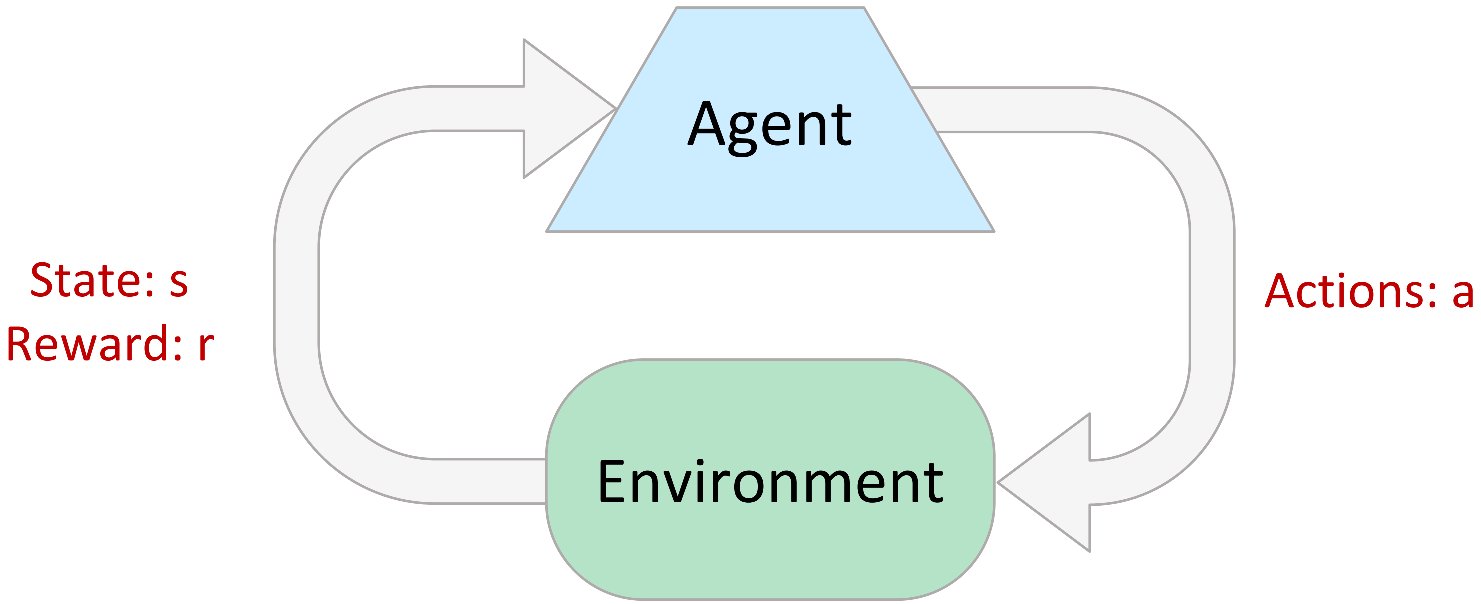
- In Reinforcement Learning (RL)
  - We still have an MDP
    - A set of states s ∈ S
    - A set of actions (per state) A
    - A transition model T(s,a,s')
    - A reward function R(s,a,s')
  - Still looking for a policy π(s)

- New twist: we don't know T, R
  - i.e. we don't know which states are good or what the actions do
- So, we must actually try out actions and states to learn

So RL can solve MDP problems when we don't know the MDP

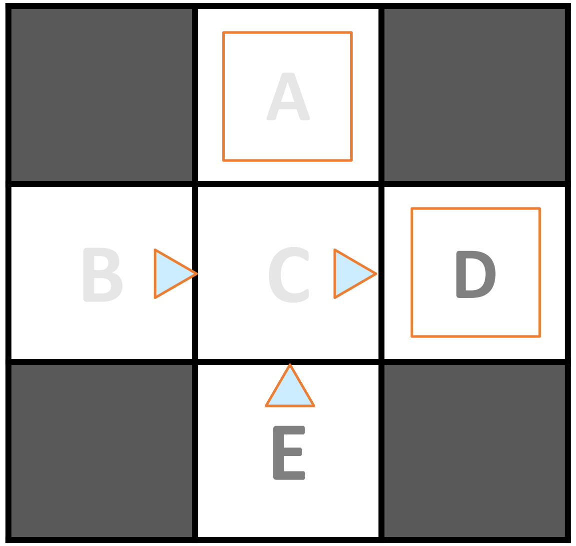
Reinforcement Learning

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



Collecting experience

- The agent collects experience/data via its interaction with the environment
- Tuples *(s,a,s',r)* are known as samples
- A collection of samples until arriving at a terminal state is known as episode



Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

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

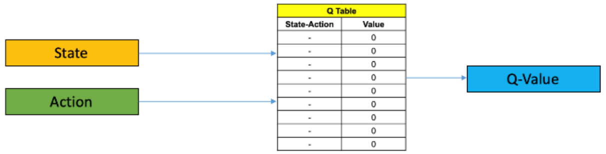
Episode 4

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

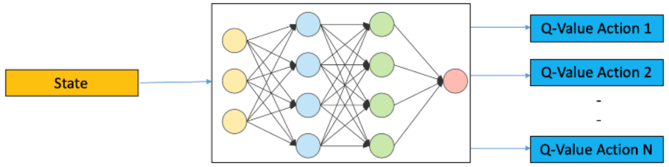
Key ideas for learning

- Online vs Offline/Batch learning
  - Learn while exploring the world, or learn from fixed batch of data
- Active vs. Passive Learning
  - Does the learner actively choose actions to gather experience? or, is a fixed policy provided?
- Model-based vs. Model-free Learning
  - Do we estimate *T(s,a,s')* and *R(s,a,s')*, or just learn values/policy directly?

- What we will (quickly ☹) cover in the next 3-4 lectures
  - Model-based learning
  - Model-free learning
    - Passive RL (direct evaluation, TD-learning)
    - Active RL (Q-learning)
  - Value-function approximation (Approximate (deep) Q-learning)



Q Learning



Deep Q Learning

Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

Model-based learning

- Model-based idea:
  - (Step 1) Learn an approximate model of *T*, *R* based on experiences/data
  - (Step 2) Solve the MDP based on the learned *T, R*

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

- **Step 2:** Solve the learned MDP
  - For example, use value iteration or policy iteration (see previous slides)

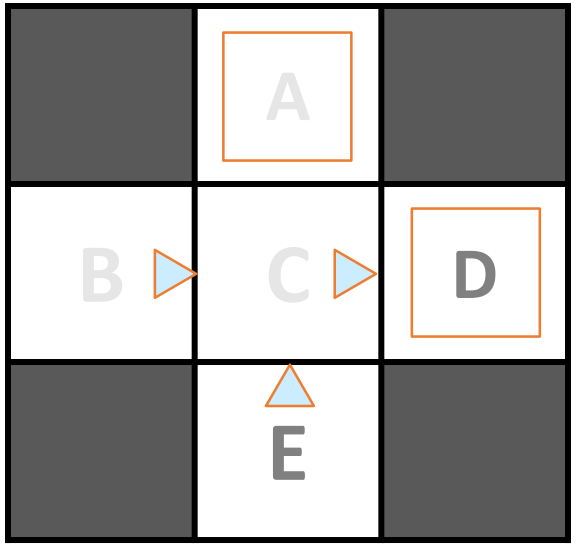
Example: Model-based learning – Step 1: Learn the empirical model

Recall that T(s,a,s')=P(s'|s,a)

T(C, east, D)=P(D|C, east)=3/4

T(C, east, A)=P(D|C, east)=1/4

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

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

Episode 4

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

Step 1:
For each q-state, count outcomes, e.g., for (C,east):
{C, east, D, -1;
C, east, D, -1;
C, east, D, -1;
C, east, A, -1;}

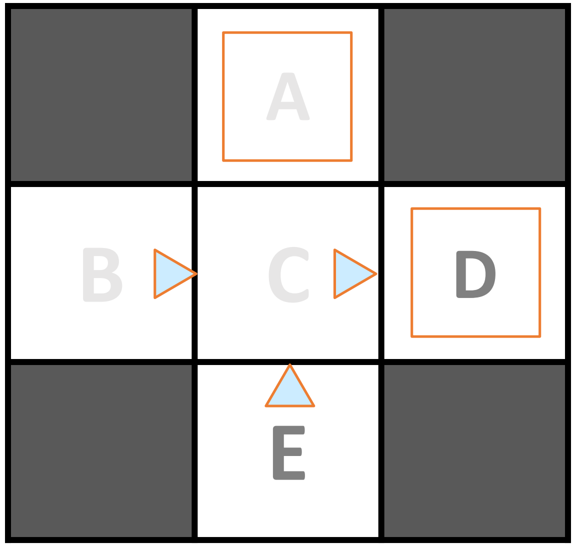
Based on these outcomes, we can compute the probability of each outcome → *T*

For each sample *(s,a,s')* discover the associated reward → *R*

Assumption: the reward is deterministic

Example: Model-based learning – Step 1: Learn the empirical model

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

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

Episode 4

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

Learned Model

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

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

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

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

Assumption: the reward is deterministic

Example: Model-based learning – Step 2: Solve the learned MDP

- As the experience increases and we collect more and more samples, our empirical models will improve:
  - The estimated transition model $\hat{T}$ will converge towards real *T*
  - new rewards will be discovered as new *(s,a,s')* tuples are explored.

- When the estimates are adequate, the training phase (Step 1) ends

- (Step 2)Based on the learned parameters, we solve the (conventional) MDP
  - using value iteration or policy iteration (see previous slides)

Model-based learning: discussion

- Model-based Idea:
  - (Step 1) Learn an approximate model of *T, R* based on experiences/data
  - (Step 2) Solve the conventional MDP based on the learned *T, R*

- Pros
  - Very simple and intuitive
  - Remarkably effective

- Cons
  - Sufficient (training) experience is required
  - Maintaining all these counts is expensive

Outline

- Introduction
- MDP formulation
- Solving the MDP
- From value-iteration to policy-iteration
- Relaxing the (full) MDP assumptions →RL
- Model-based learning
- Things you should know from this lecture & reading material

Overview and Reading

- Overview
  - RL basics
  - MDP formulation
  - Bellman equations
  - Value iteration
  - Policy extraction
  - Policy evaluation
  - Policy iteration
- Reading
  - Chapter 16& 23, AI book, 4th edition
  - RL Book, Barto and Sutton, 2nd edition
  - Introduction to Reinforcement Learning with David Silver (DeepMind)
  - Stanford CS234: Reinforcement Learning with Emma Brunskill

Hands on experience

- Small programming exercise
  - For a (small) grid-world example similar to our toy example implement from scratch
    - Value iteration
    - Policy extraction
    - Policy evaluation (try out different policies, e.g., go always in one direction {N,S,W,E}, act randomly etc)
    - Policy iteration
  - Assuming now that you don't know the *R, T* components, implement the model-based RL version

- Familiarize yourself with OpenAI Gym
  - Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.
- We will release a 3rd project (CartPole balancing problem, most probably)
  - Look at this presentation of the CartPole problem (states, actions, rewards, …)

Thank you

Questions/Feedback/Wishes?

Acknowledgements

- The slides are based on
  - CS 188 | Introduction to Artificial Intelligence, Berkeley
  - Artificial Intelligence: A modern approach (Russel and Norvig), 4th edition