COL333/671: Introduction to AI

Semester I, 2021

Markov Decision Processes

Rohan Paul

Outline

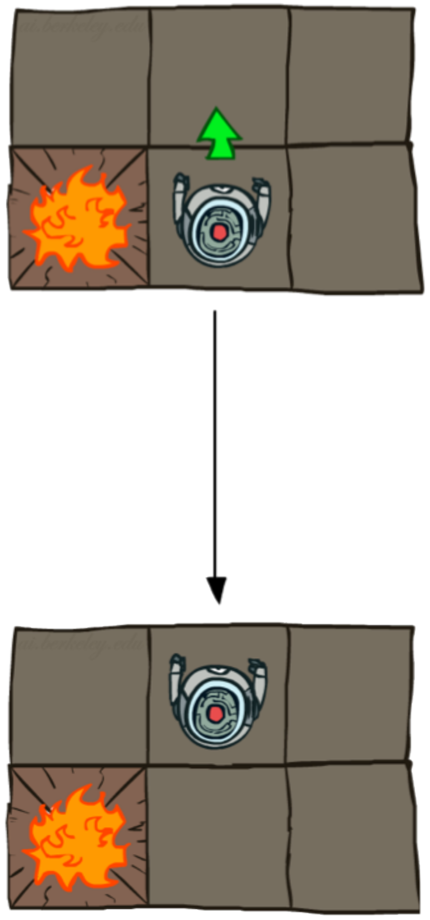
- Last Class
  - Utilities and Probabilities
- This Class
  - Markov Decision Processes
- Reference Material
  - Please follow the notes as the primary reference on this topic. Supplementary reading on topics covered in class from AIMA Ch 17 sections 17.1 – 17.3.

Acknowledgement

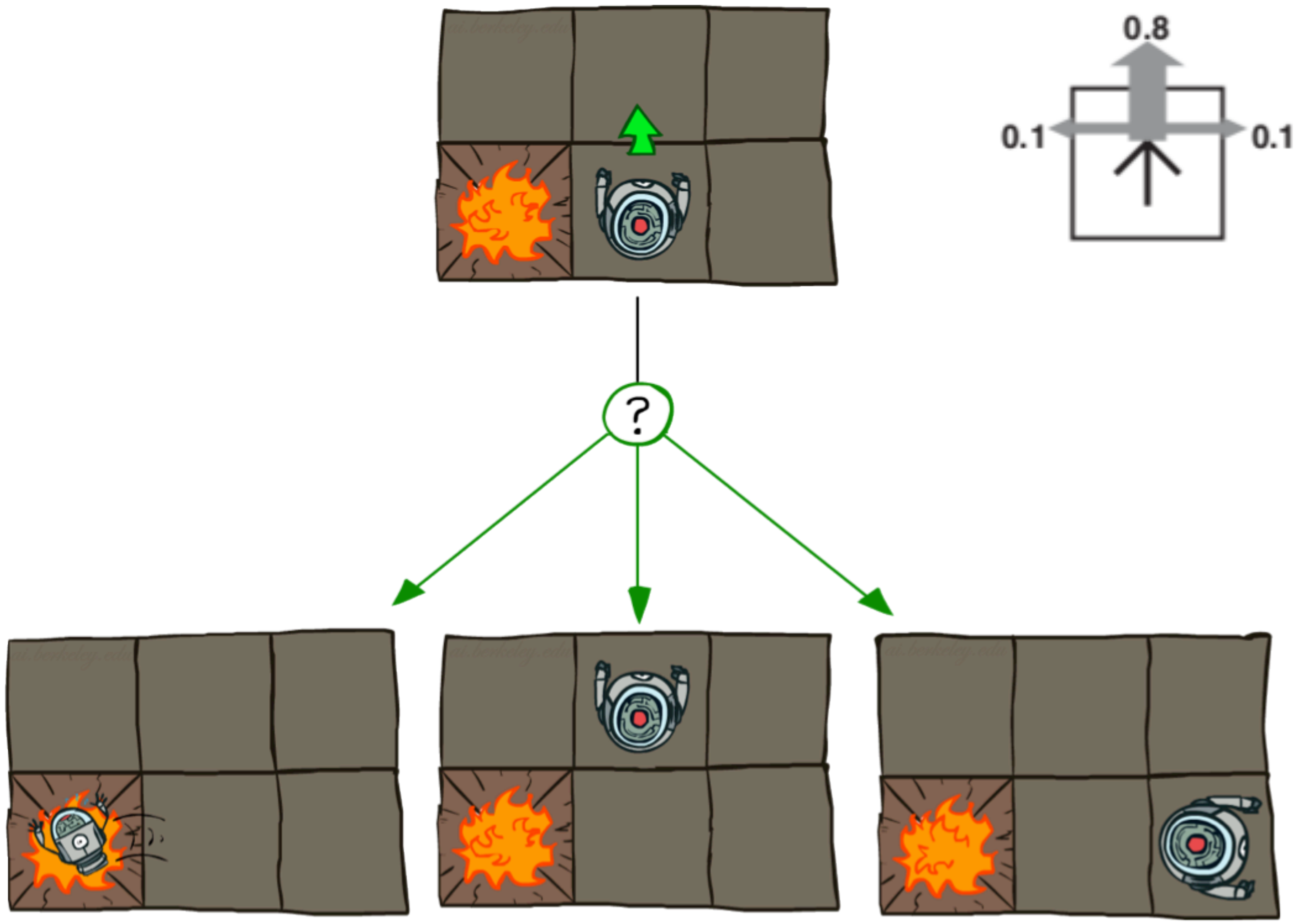
These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Doina Precup, Dorsa Sadigh, Percy Liang, Mausam, Dan Klein, Anca Dragan, Nicholas Roy, Emilio Frazzoli and others.

Deterministic vs. Stochastic Actions

Deterministic Action Outcomes



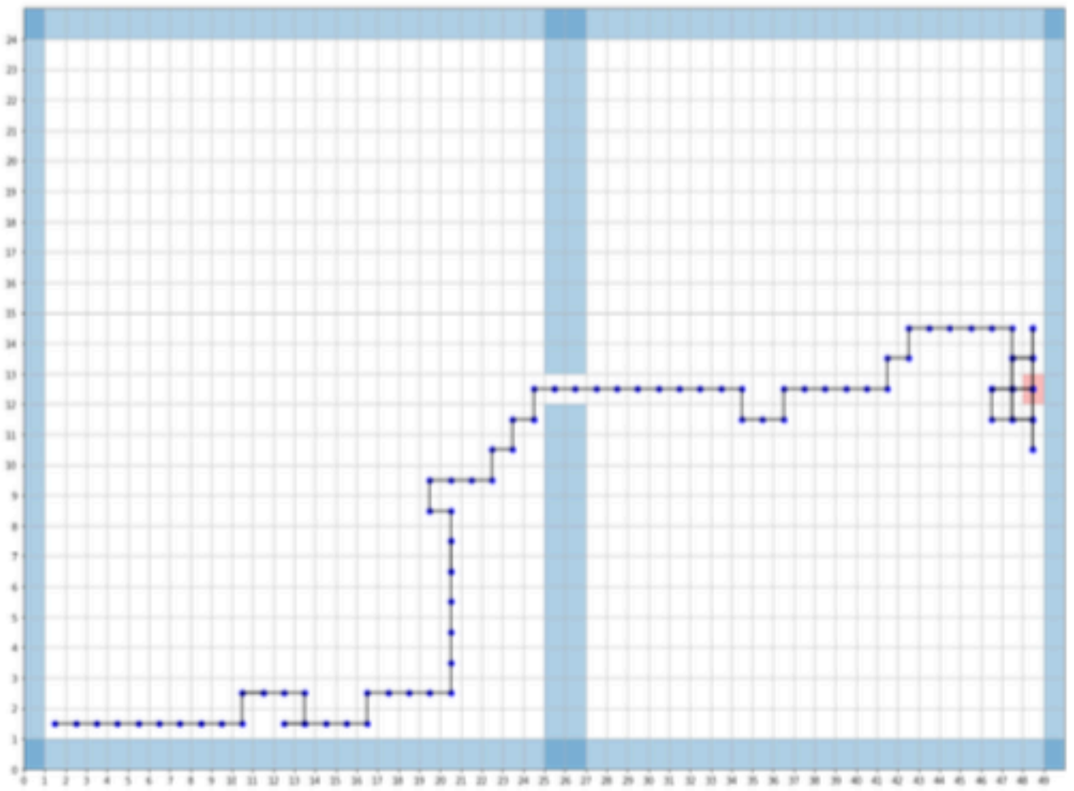
Stochastic Action Outcomes



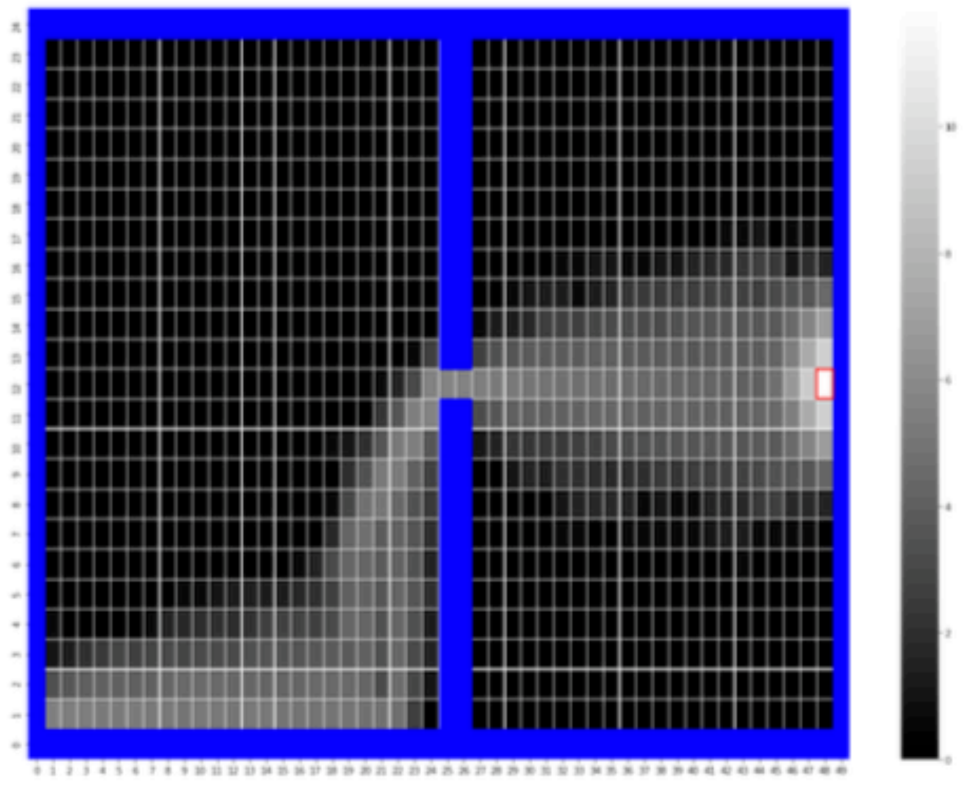
Need to plan for contingencies

Example: Sample paths through an MDP

A sample path through the MDP

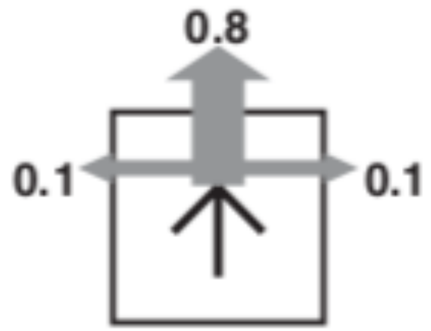


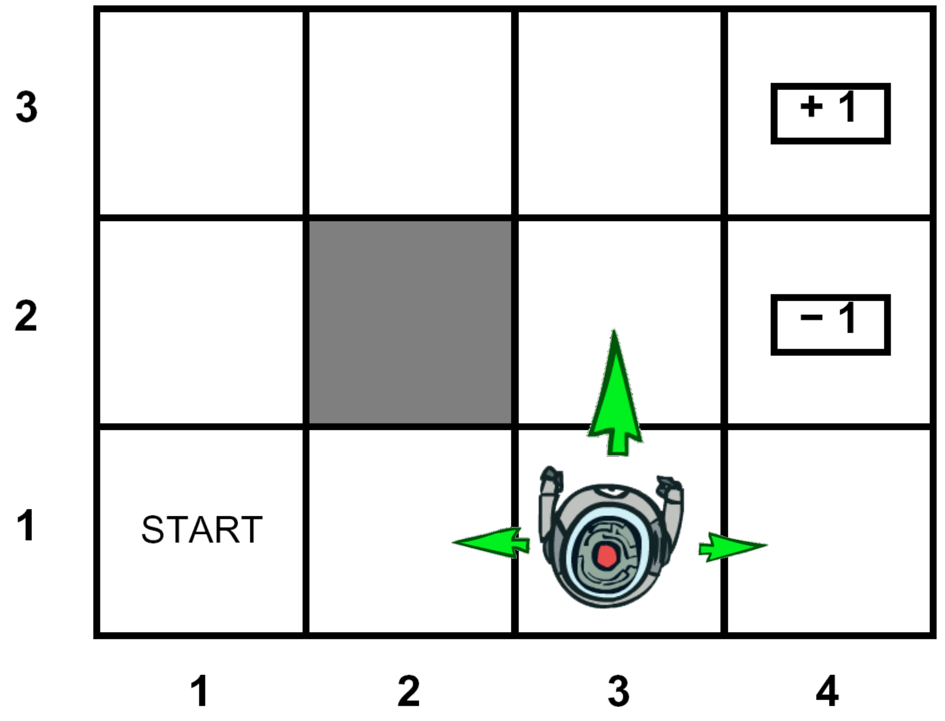
State visitations after multiple runs



Grid World Example

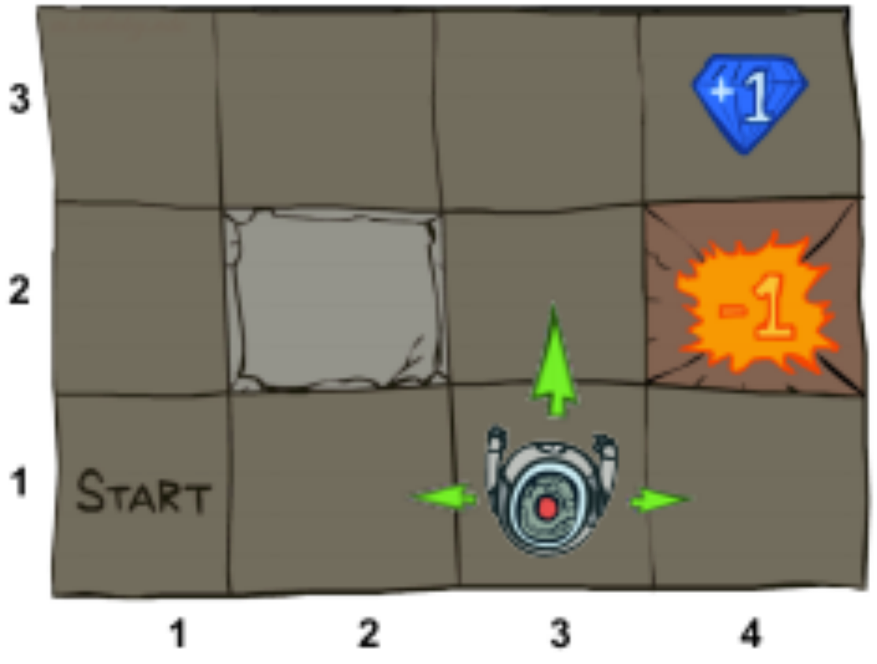
- Actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small "living" reward each step (can be negative)
  - Large rewards come at the end (negative or positive)
- Overall Goal: maximize sum of rewards
  - Fundamentally a sequential decision-making problem.
  - Taking an action now can have an impact later.





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



$T(s_{11}, E, \ldots$

$\ldots$

$T(s_{31}, N, s_{11}) = 0$

$\ldots$

$T(s_{31}, N, s_{32}) = 0.8$

$T(s_{31}, N, s_{21}) = 0.1$

$T(s_{31}, N, s_{41}) = 0.1$

$\ldots$

T is a Big Table!

11 X 4 x 11 = 484 entries

For now, we give this as input to the agent

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')

…

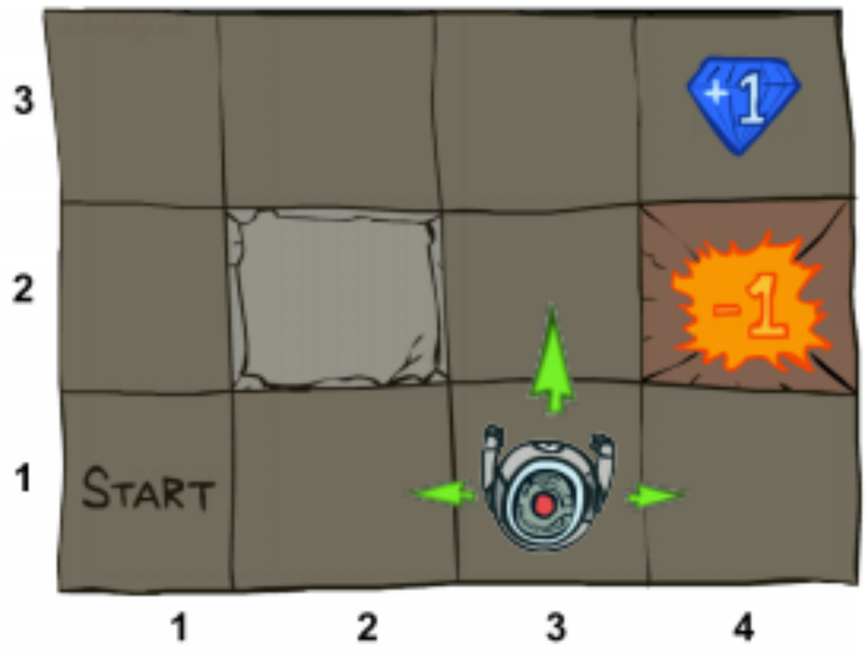
$R(s_{32}, N, s_{33}) = -0.01$ ← ***Cost of breathing***

…

$R(s_{32}, N, s_{42}) = -1.01$

…

$R(s_{33}, E, s_{43}) = 0.99$



R is also a Big Table!

For now, we also give this to the agent

Markov Decision Processes

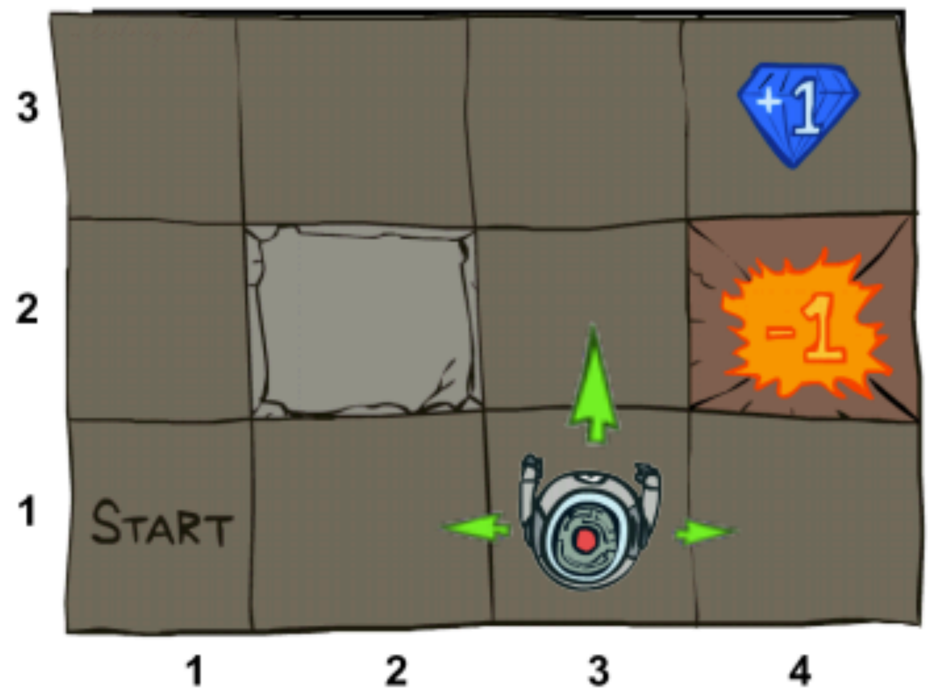
- An MDP is defined by:
  - A set of states s in S
  - A set of actions a in A
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s'| s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')

...

$R(s_{33}) = -0.01$

$R(s_{42}) = -1.01$

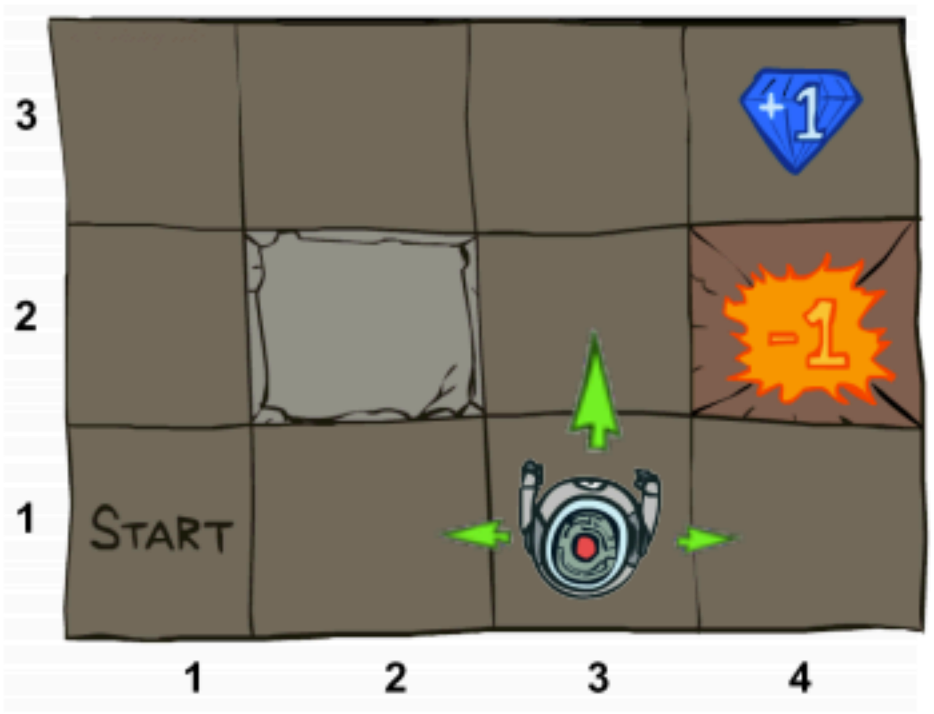
$R(s_{43}) = 0.99$



Note: two notations are followed in literature. One in which rewards are associated with states and in the other rewards are associated with state transitions. Both the notations are equivalent and accepted.

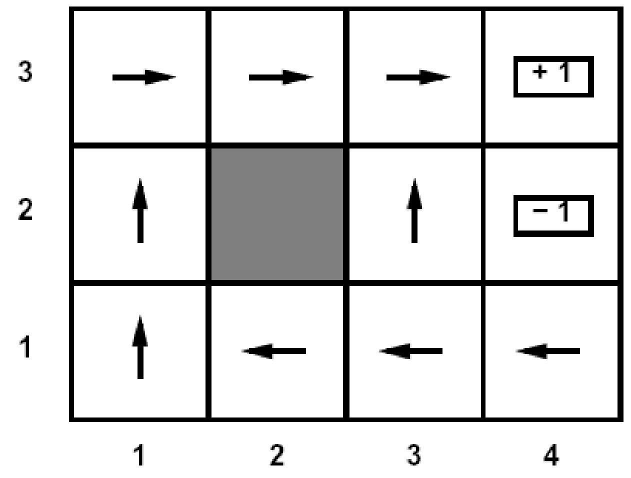
Markov Decision Processes

- 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

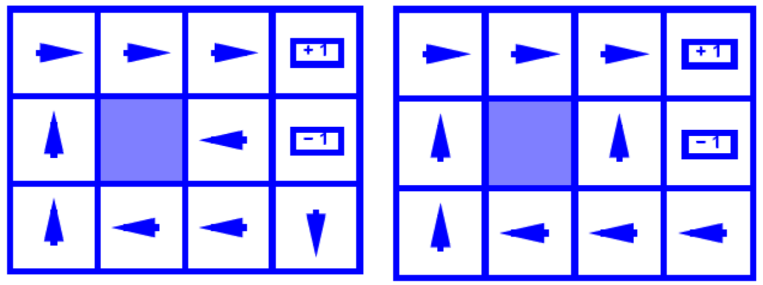


Policies

- Deterministic single-agent search problems
  - We determined the optimal plan, or sequence of actions, from start to a goal
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - **An optimal policy is one that maximizes the expected utility if followed**
  - The agent arrives at a state and looks up the action according to the policy.
- Note: there can be many policies, we are to determine the optimal one.



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



There can be other policies that prescribe different actions in a state.

Markov Assumption in MDPs

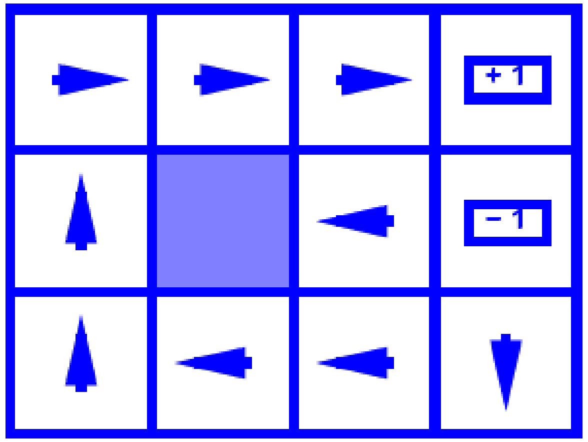
- "Markov"
  - Given the present state, the future and the past are independent
- For Markov decision processes
  - "Markov" means action outcomes depend only on the current state
  - The next state depends on the action and the current state.
  - The past actions taken the past states encountered do not affect the next state.

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

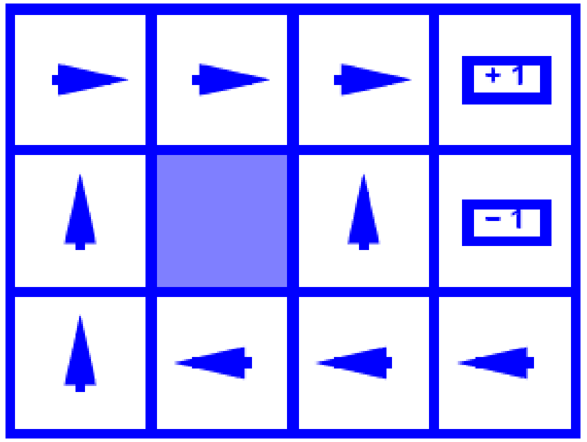
$$= \quad P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

Optimal policies for different living rewards

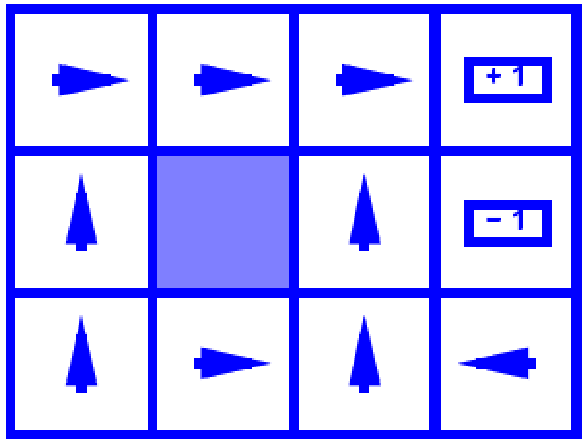
- Interpret reward as the cost of breathing (living reward).
- The value of R(s) balances the risk and reward that the agent takes.



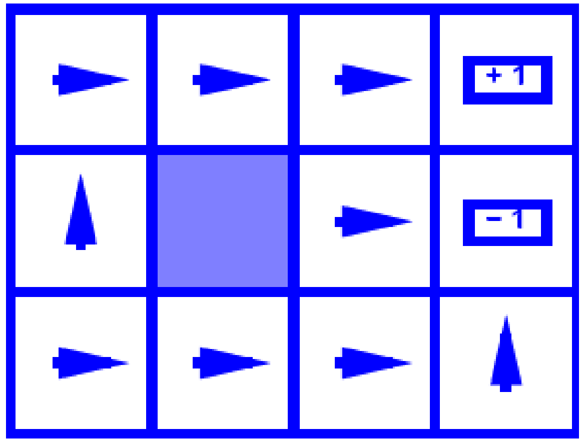
R(s) = -0.01



R(s) = -0.03



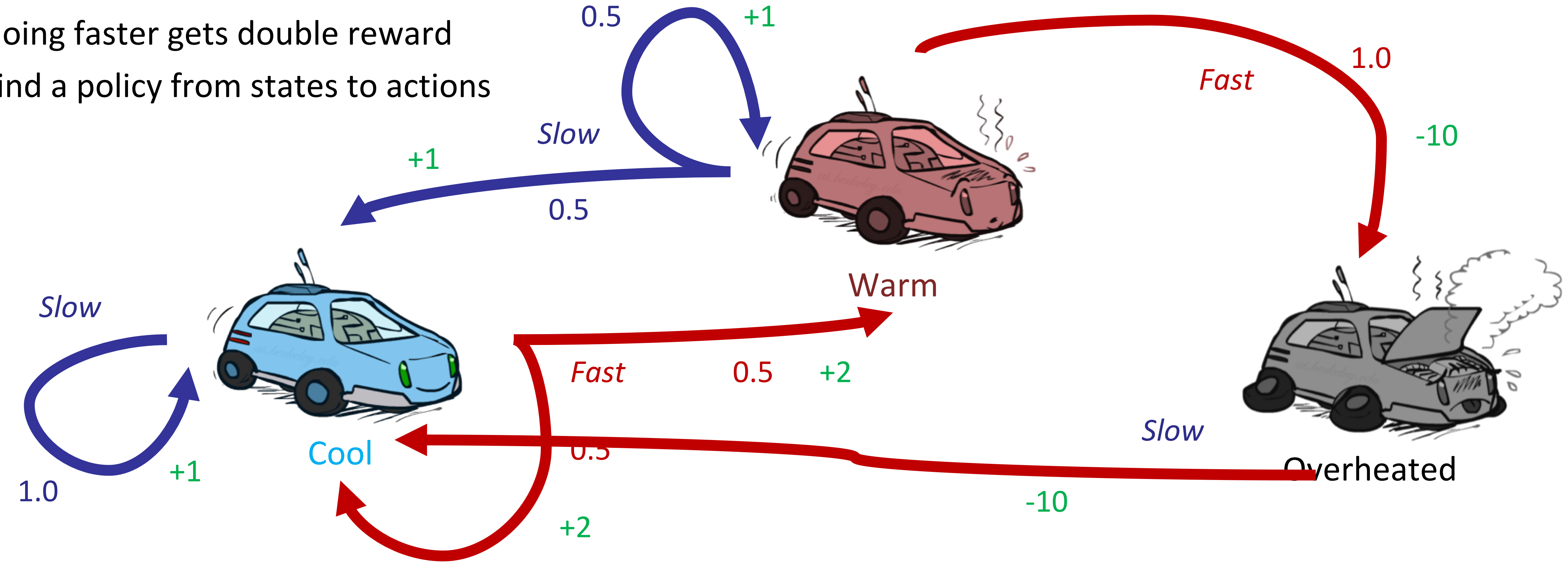
R(s) = -0.4



R(s) = -2.0

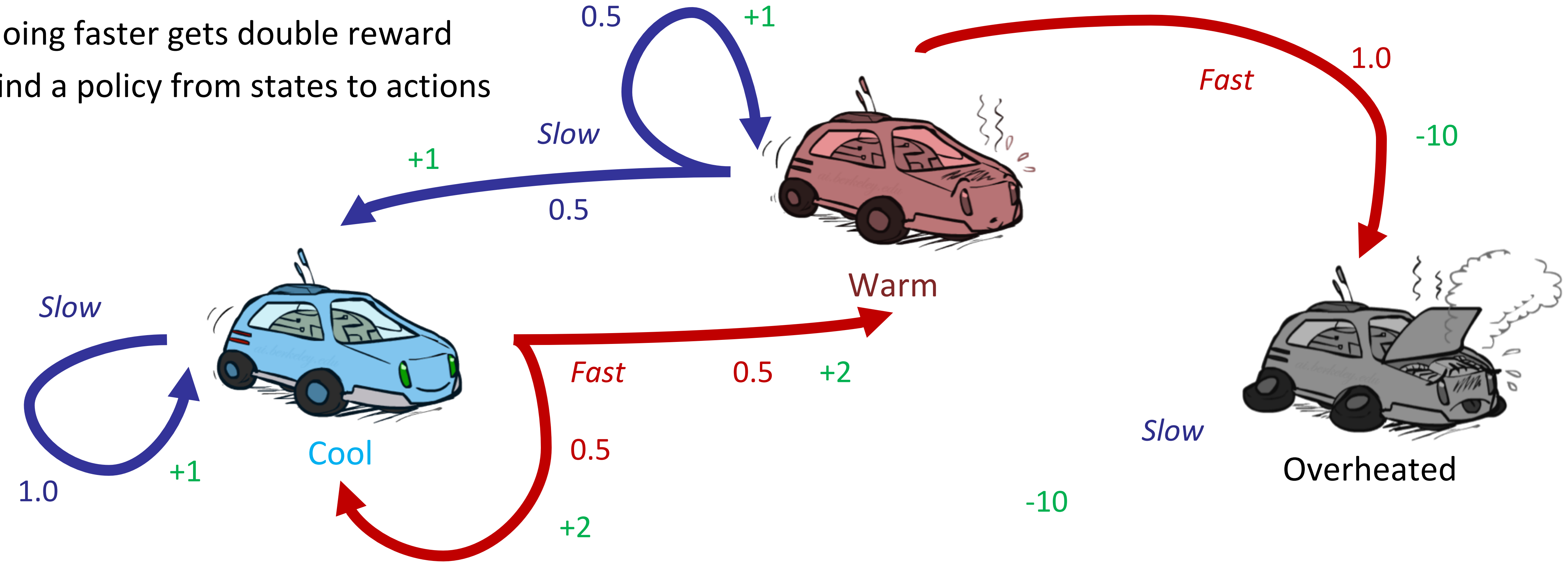
Example: Racing Car

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



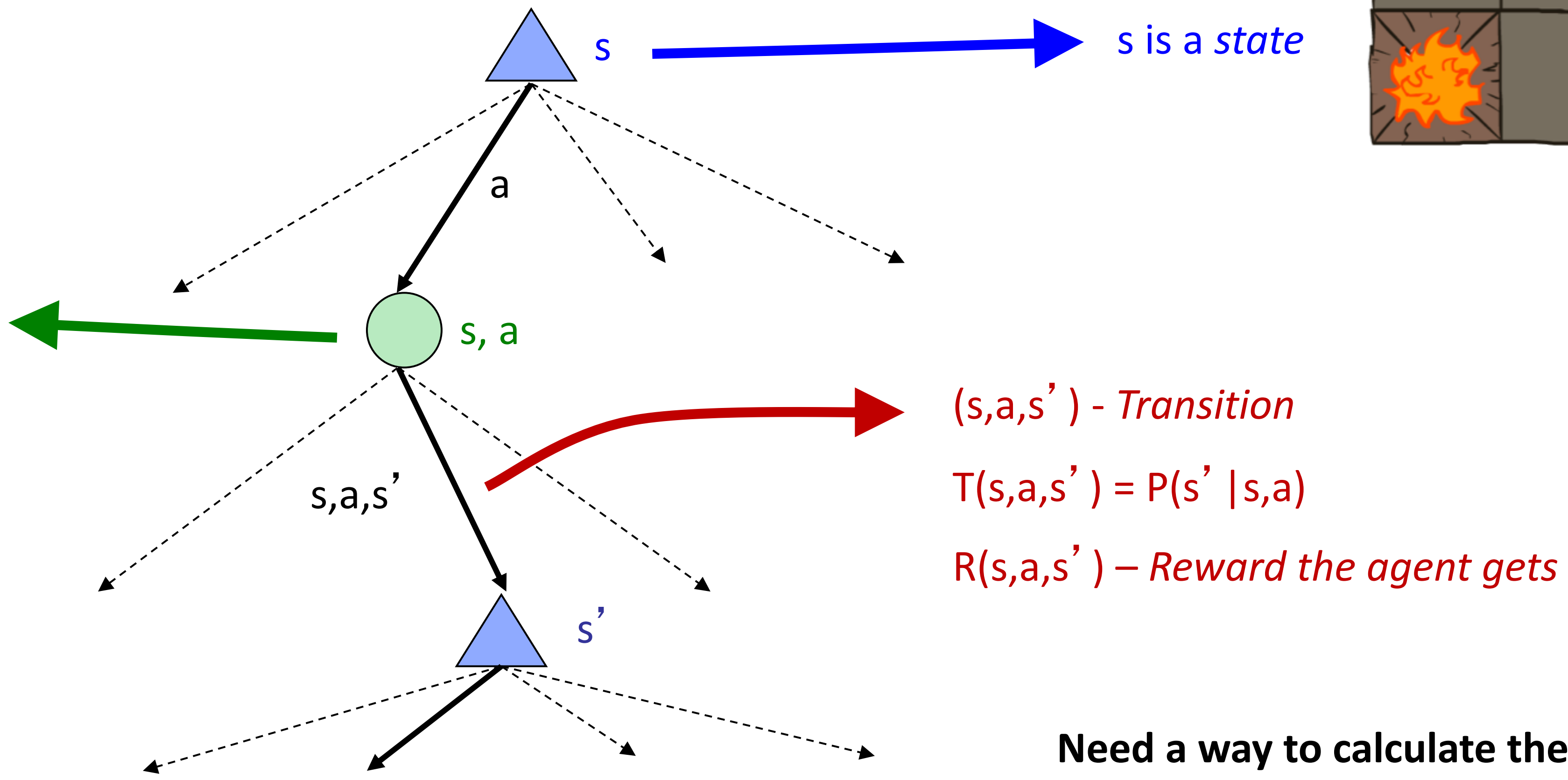
Example: Racing Car

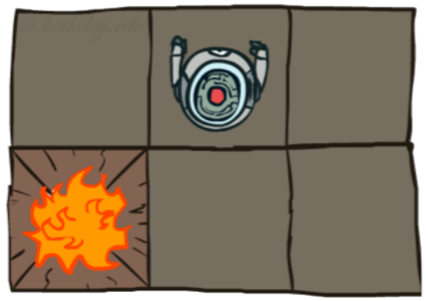
- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated (terminal state)
- Two actions: *Slow*, *Fast*
- Going faster gets double reward
- Find a policy from states to actions



MDP as a Search Tree

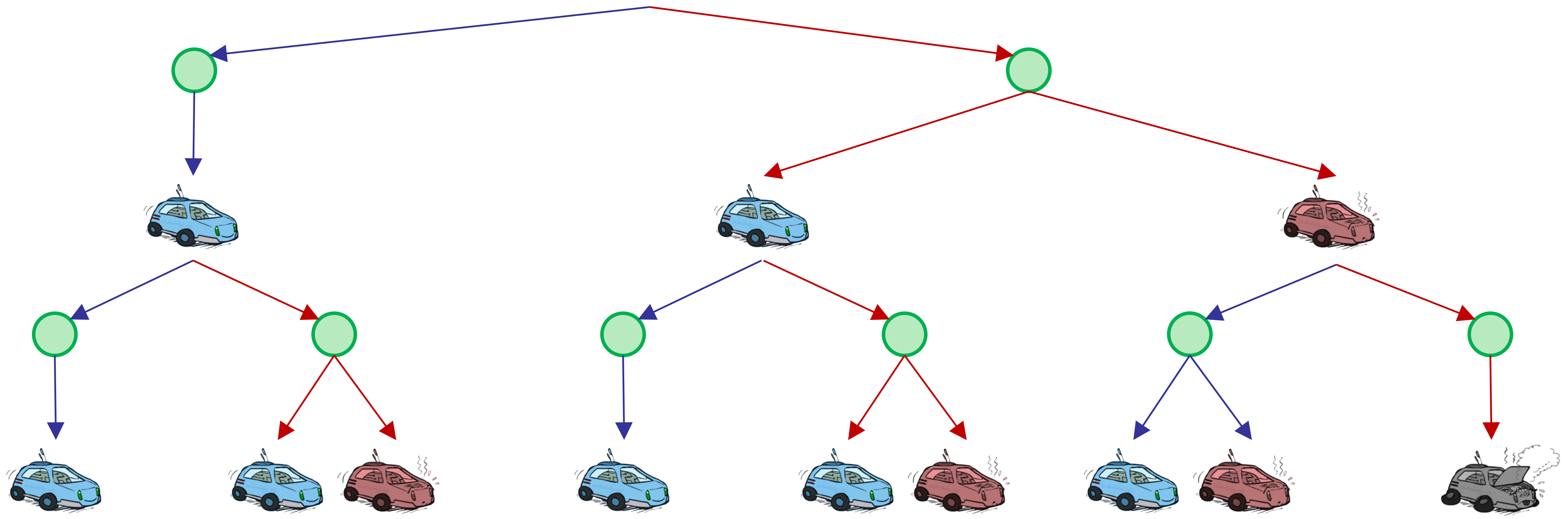
Target: Need to find the optimal policy. The one that maximizes the expected utility if followed.





Need a way to calculate the utility of a sequence of states!

Example: Racing Car



Visualizing an MDP as an Expectimax tree. Imagining the consequences of actions into the future.

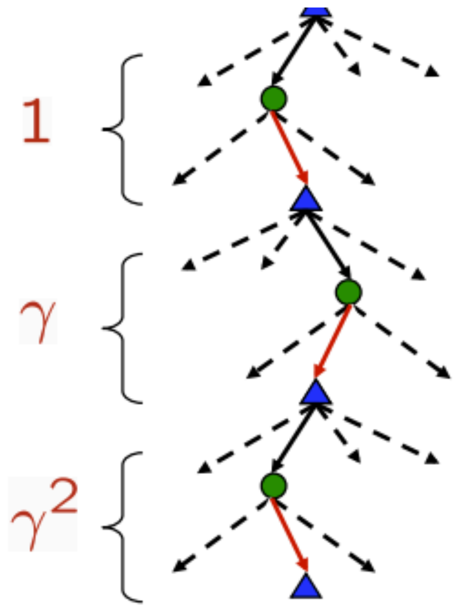
Utility of Reward Sequences

What preferences should an agent have over reward sequences?

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

Maximize the sum of rewards
Prefer rewards now to rewards later - 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])



Assigning Utilities to Reward Sequences

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

Discounting appears to be a good model for both animal and human preferences over time. Computationally, helps us converge utilities of infinite sequences.

With discounted rewards, the utility of an infinite sequence is finite.

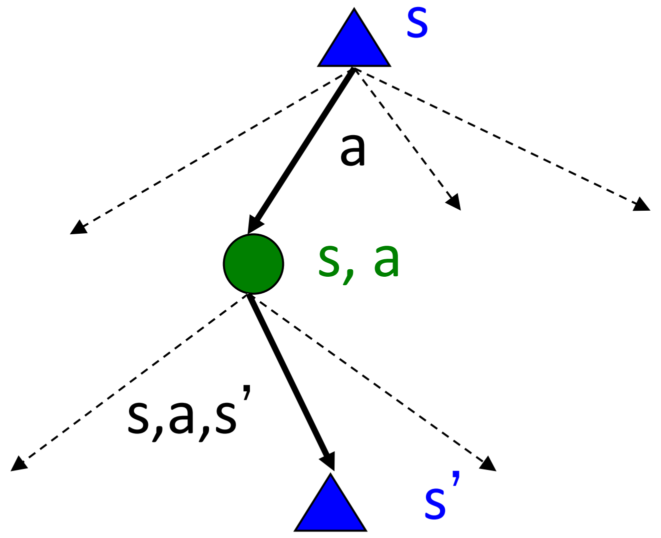
$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

MDP Formulation

- **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:**
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

- ***Next: How to solve the MDP?***
  - *How to determine the optimal policy?*

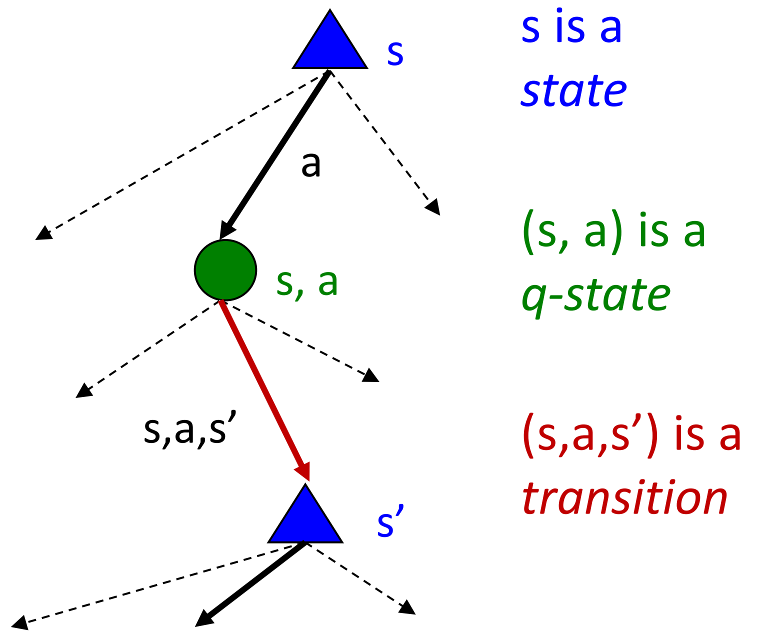


Optimal Quantities

- **The value (utility) of a state s:**
  $V^*(s)$ = expected utility starting in s and acting optimally

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

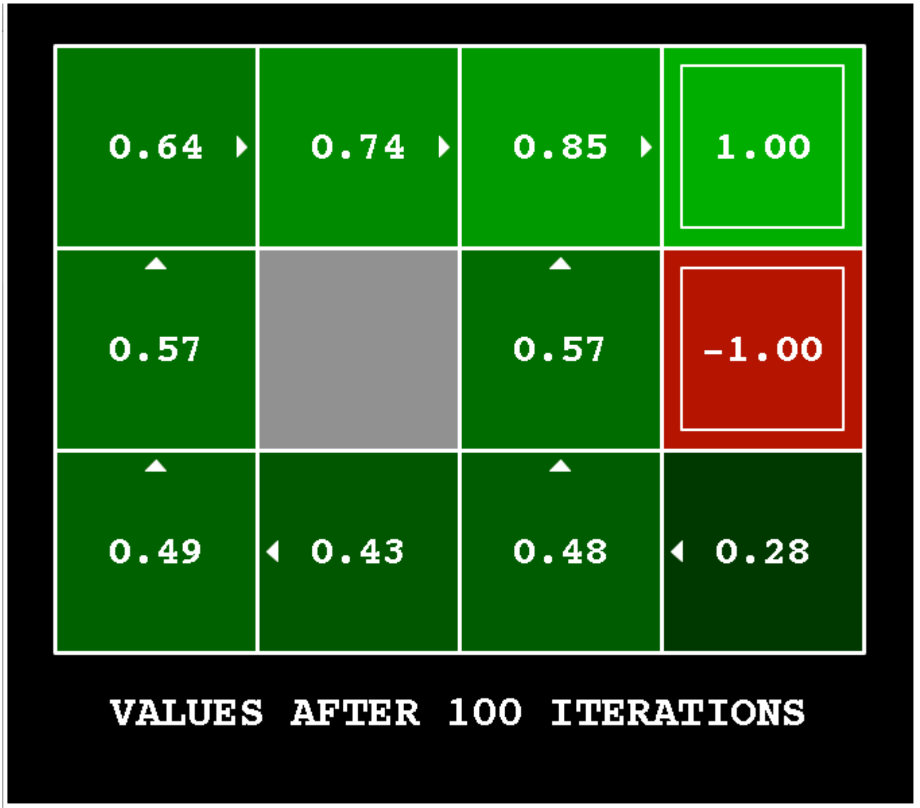
- **The optimal policy:**
  $\pi^*(s)$ = optimal action from state s

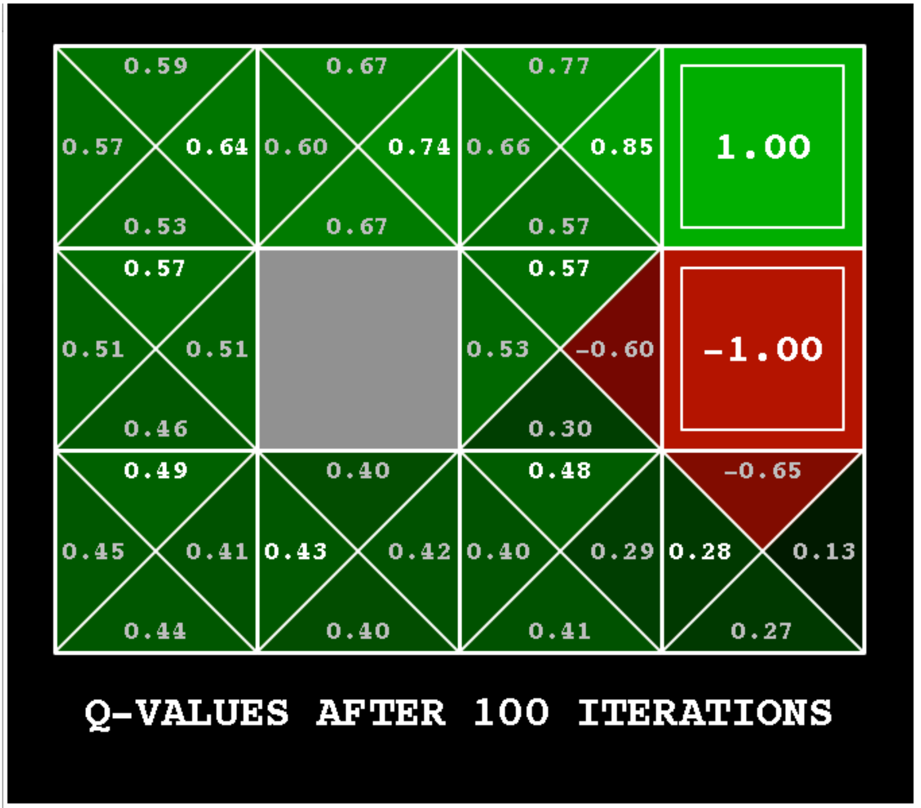


Value Function Example

Value (utility) of states V(s) for all states

Value (utility) of a q-states Q(s,a) for all states and all actions at each state.





Noise = 0.2
Discount = 0.9
Living reward = 0

Value of States

- **Values of states are related to each other.**
- **Fundamental Operation**
  - Expected utility under optimal action for this state. What is the best we can do from this state?
  - Average sum of (discounted) rewards

- **Recursive definition of value:**

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

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

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

Bellman Equations

- Definition of **"optimal"** utility gives a simple **one-step** lookahead relationship amongst optimal values.

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

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

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

- The utility of a state is the **immediate reward** for that state plus the **expected discounted utility of the next state** assuming that the agent is acting **optimally**.

Value Iteration

- Calculate the **utility of each state** and then **use the state utility** to **select an optimal action** in each state.

- Bellman equations **characterize** the optimal values:

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

- Value iteration **computes** them:

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

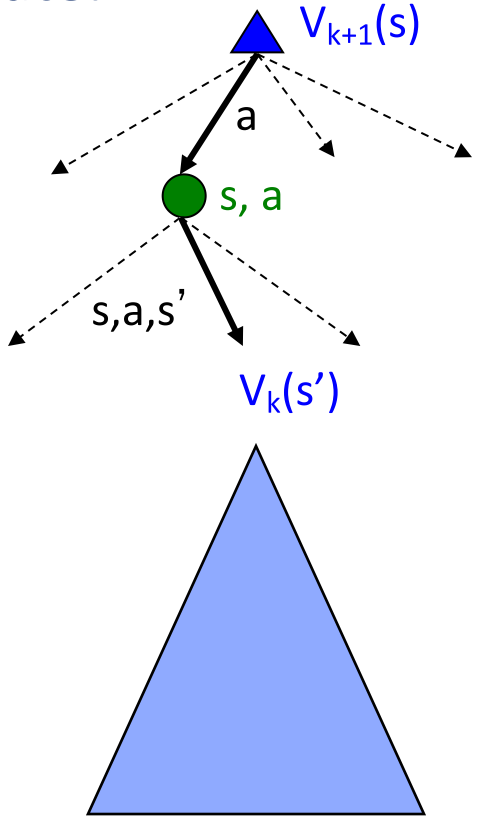
- Value iteration is a **fixed-point solution** method

Value Iteration Algorithm

- **Start with $V_0(s) = 0$**

- **Given vector of $V_k(s)$ values, do one ply of expectimax from each state:**

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



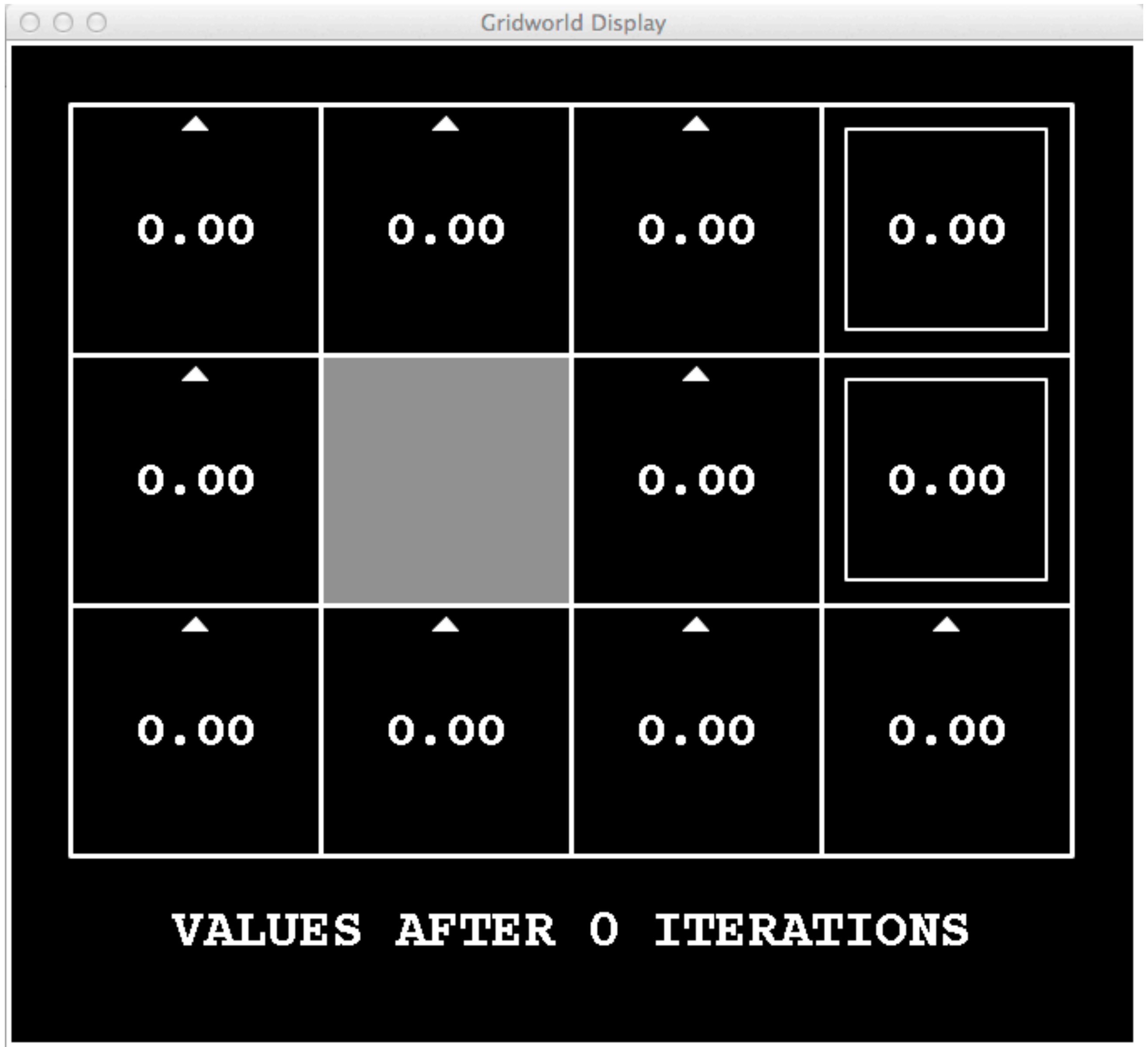
- **Repeat until convergence**
  - Determine by looking at the max. change in utility of any state in an iteration.

- Complexity of each iteration: **$O(S^2A)$**

- Theorem: will converge to unique optimal values
  - the start state does not matter

Value Iteration

k=0



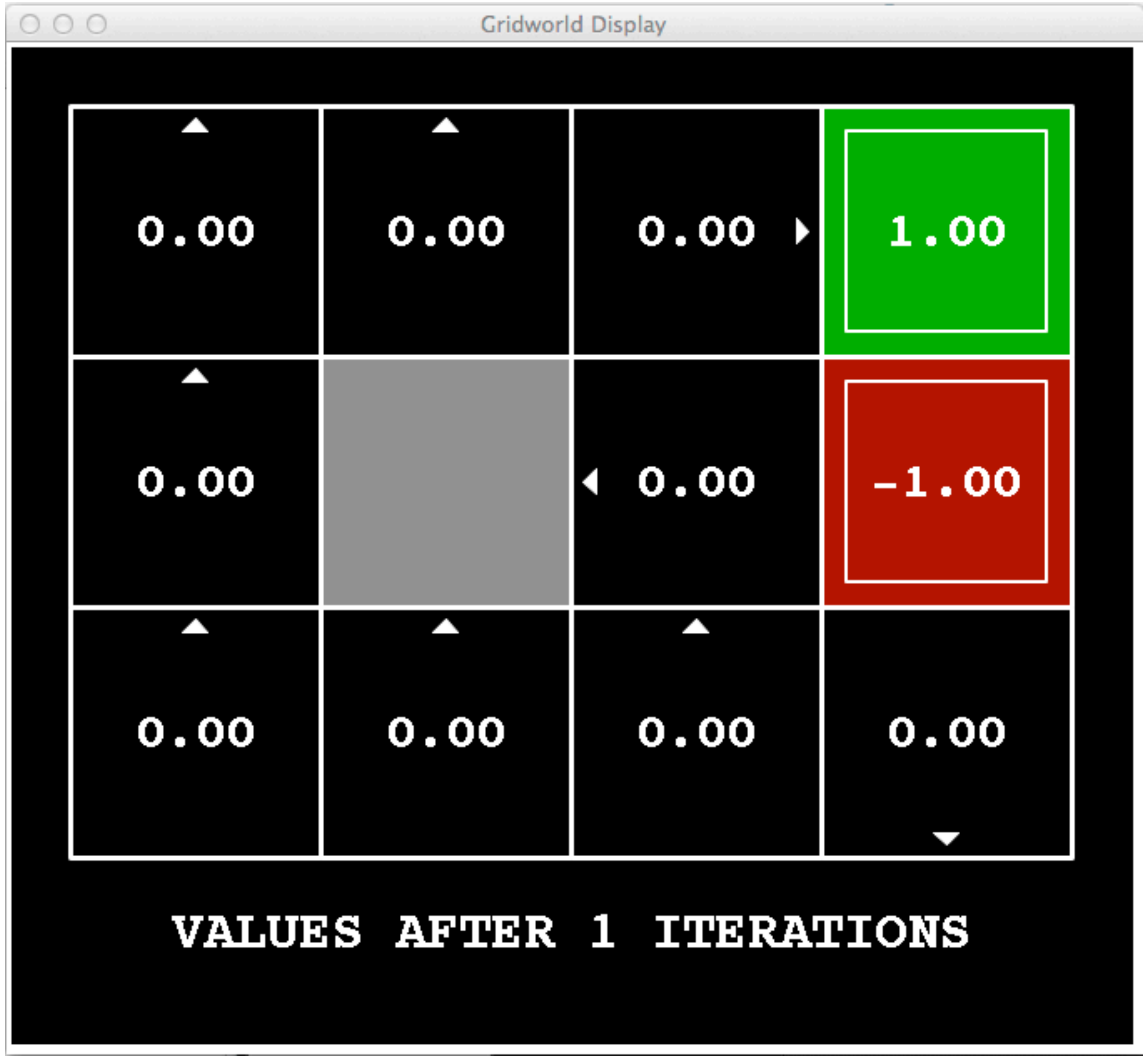
0.00	0.00	0.00	0.00
0.00		0.00	0.00
0.00	0.00	0.00	0.00

VALUES AFTER 0 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=1



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

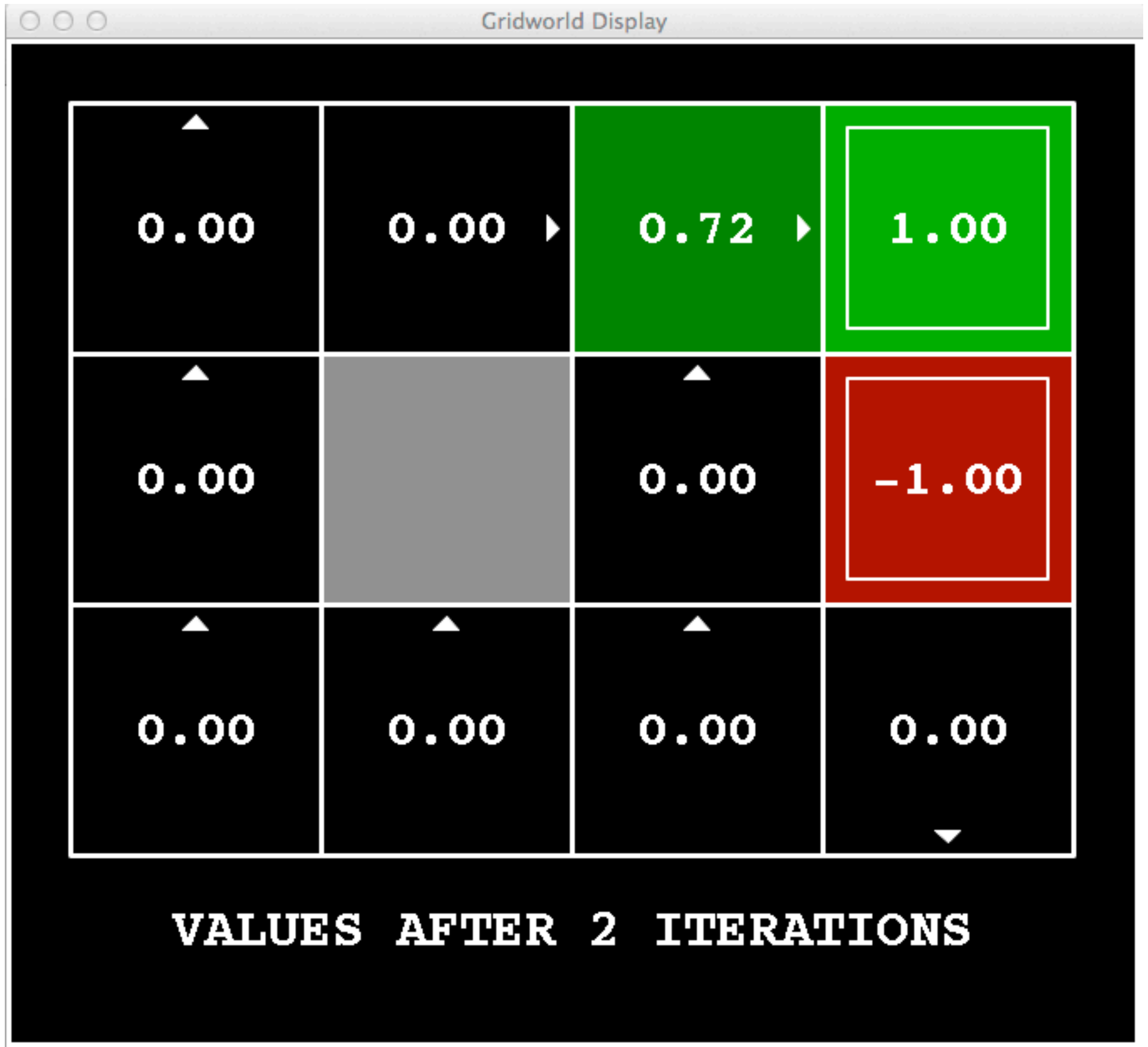
VALUES AFTER 1 ITERATIONS

In the first iteration the terminal states reflect the reward.

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=2



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

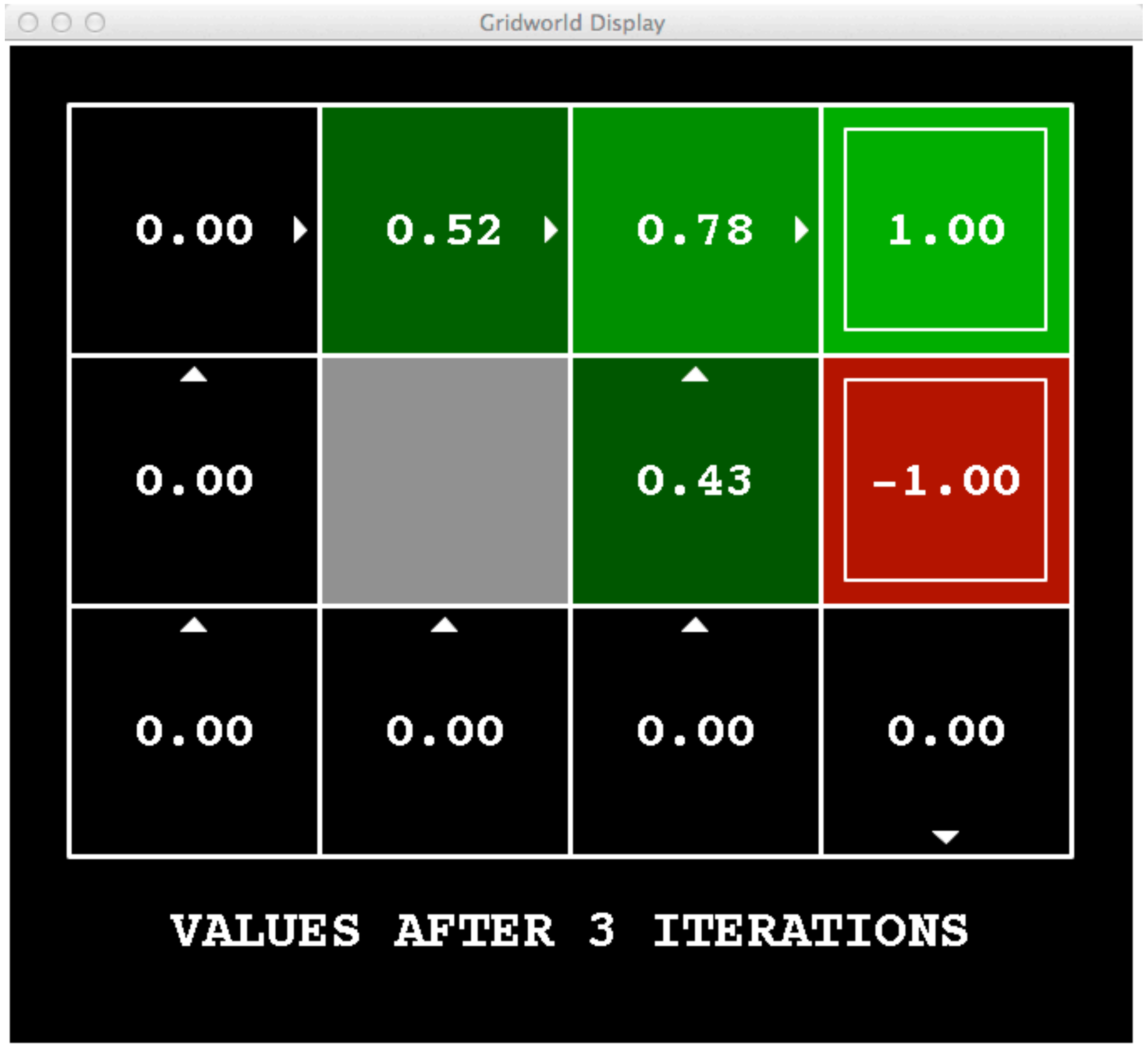
VALUES AFTER 2 ITERATIONS

Adjacent states start to get updated.

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=3



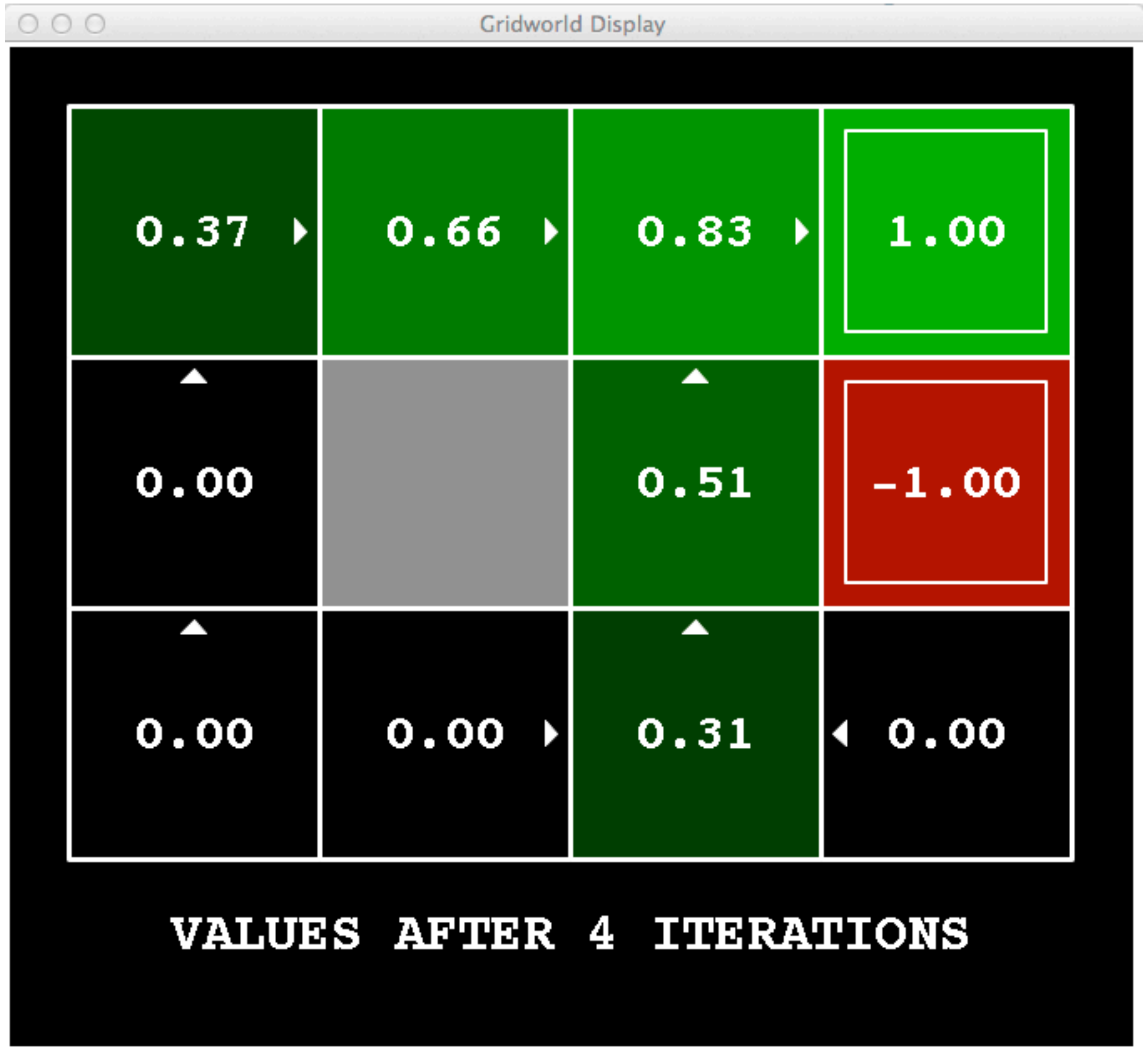
0.00 ▶	0.52 ▶	0.78 ▶	1.00
▲ 0.00		▲ 0.43	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 3 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=4



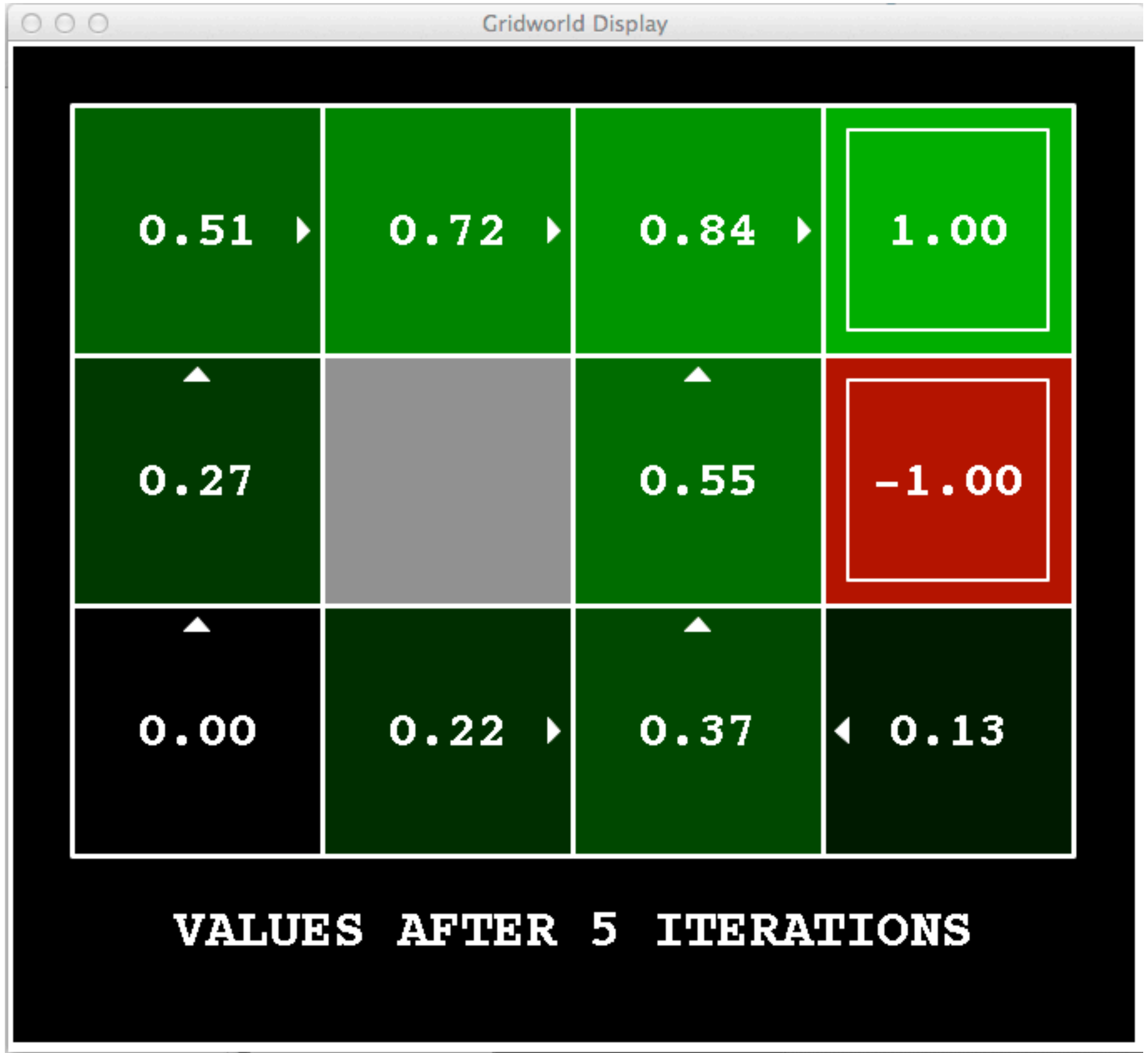
0.37 ▸	0.66 ▸	0.83 ▸	1.00
▴ 0.00		▴ 0.51	-1.00
▴ 0.00	0.00 ▸	▴ 0.31	◂ 0.00

VALUES AFTER 4 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=5



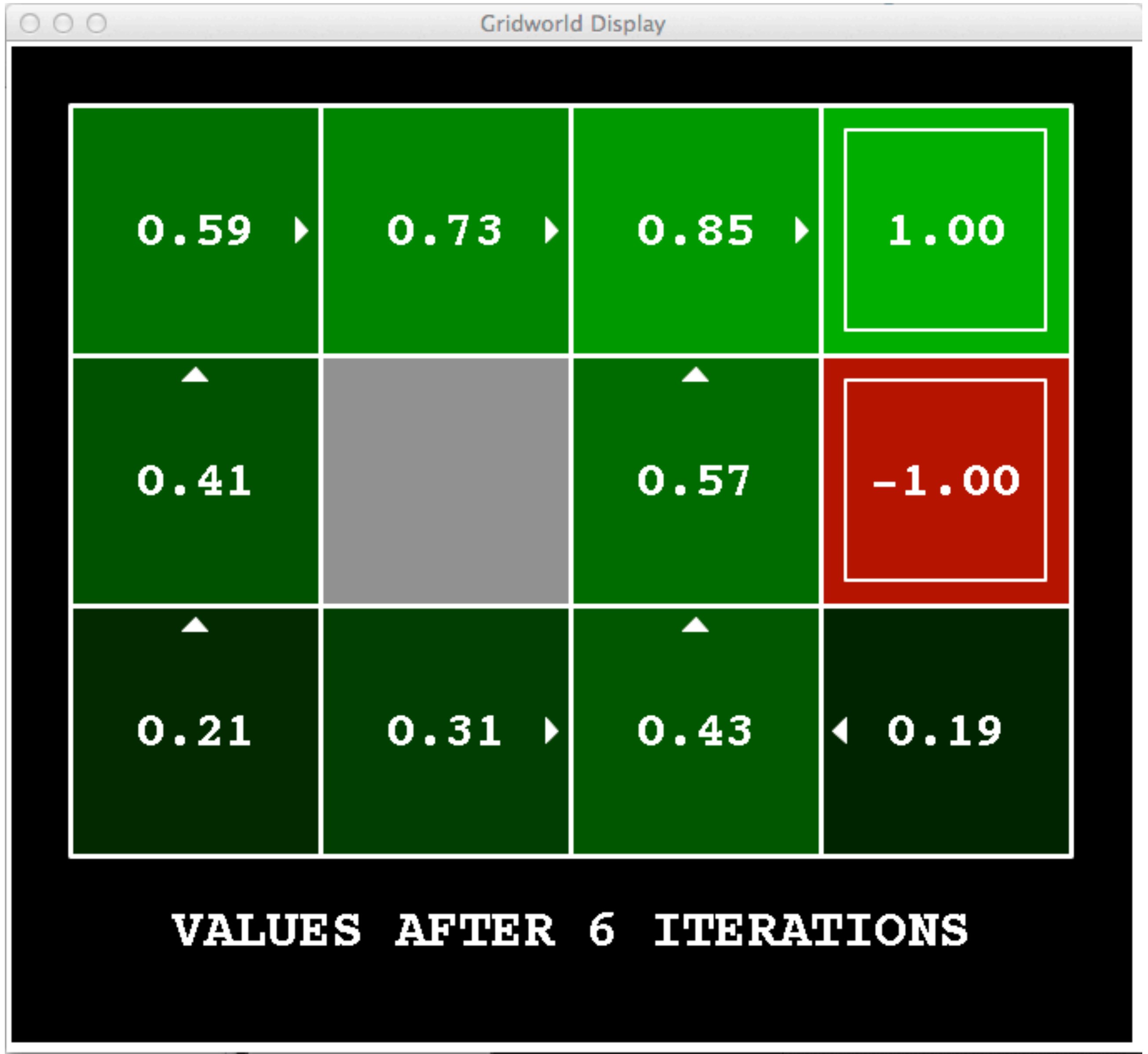
0.51 ▶	0.72 ▶	0.84 ▶	1.00
▲ 0.27		▲ 0.55	-1.00
▲ 0.00	0.22 ▶	▲ 0.37	◀ 0.13

VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=6



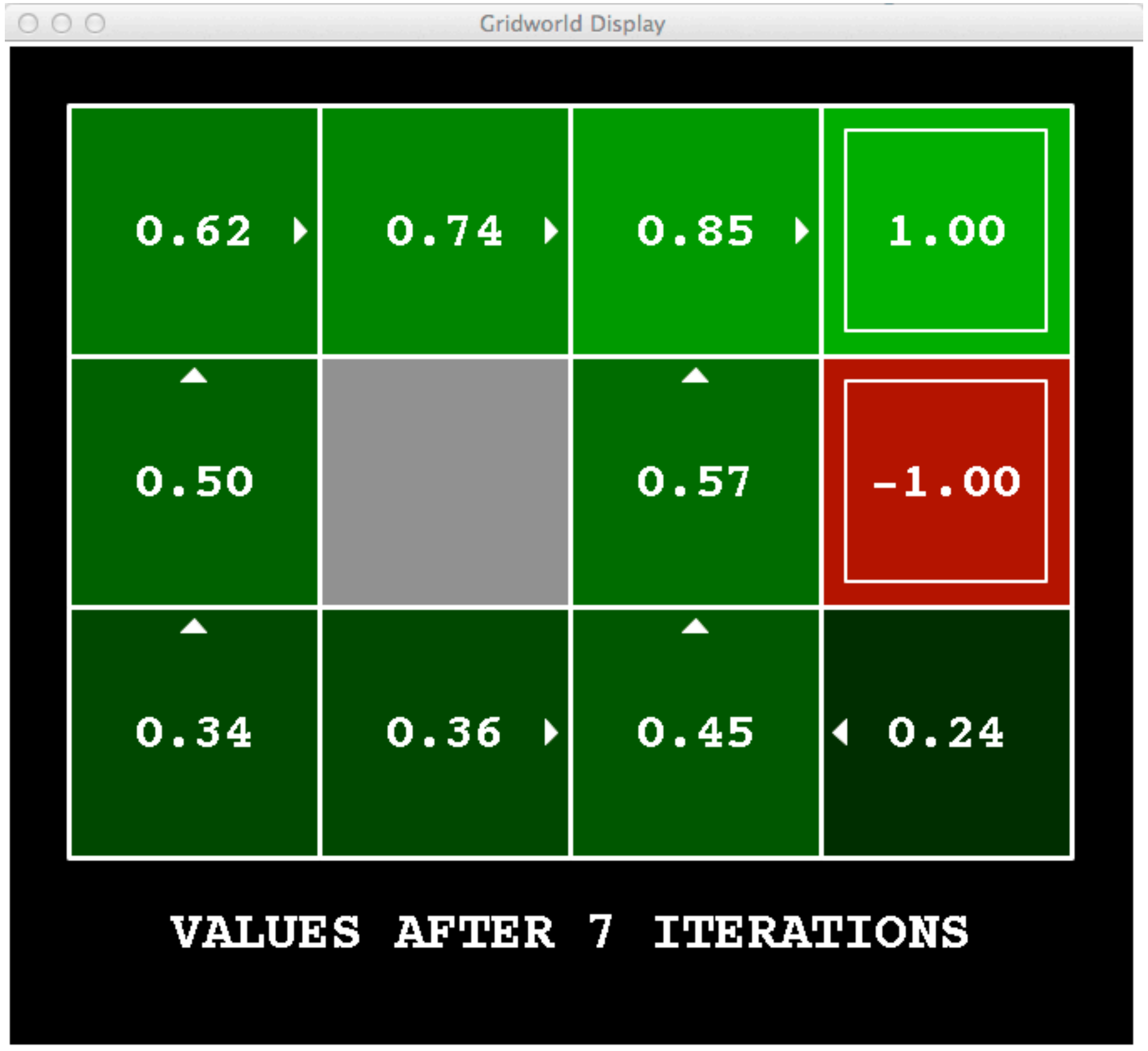
0.59 ▶	0.73 ▶	0.85 ▶	1.00
▲ 0.41		▲ 0.57	-1.00
▲ 0.21	0.31 ▶	▲ 0.43	◀ 0.19

VALUES AFTER 6 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

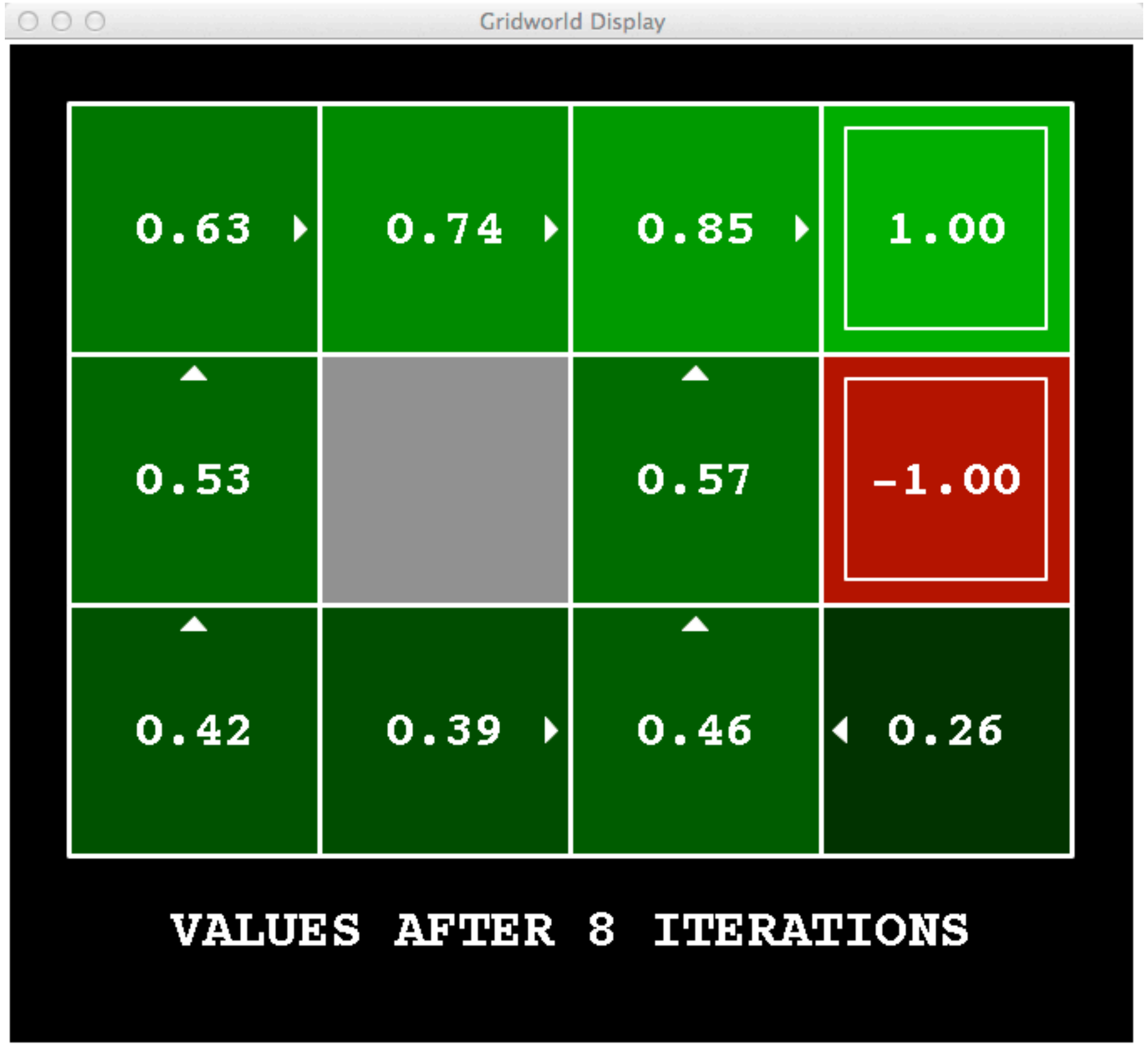
k=7



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=8



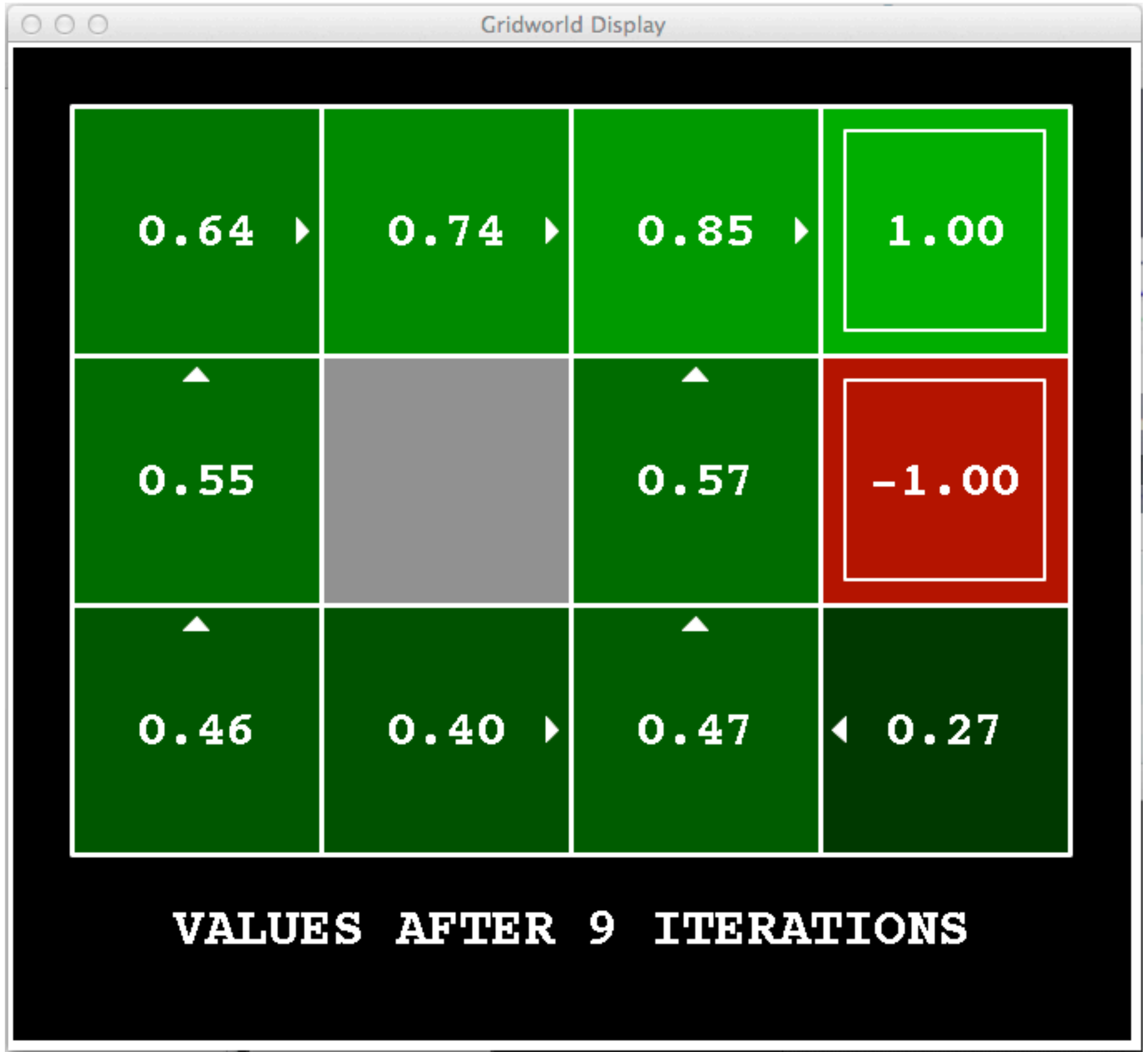
0.63 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.53		▲ 0.57	-1.00
▲ 0.42	0.39 ▶	▲ 0.46	◀ 0.26

VALUES AFTER 8 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=9



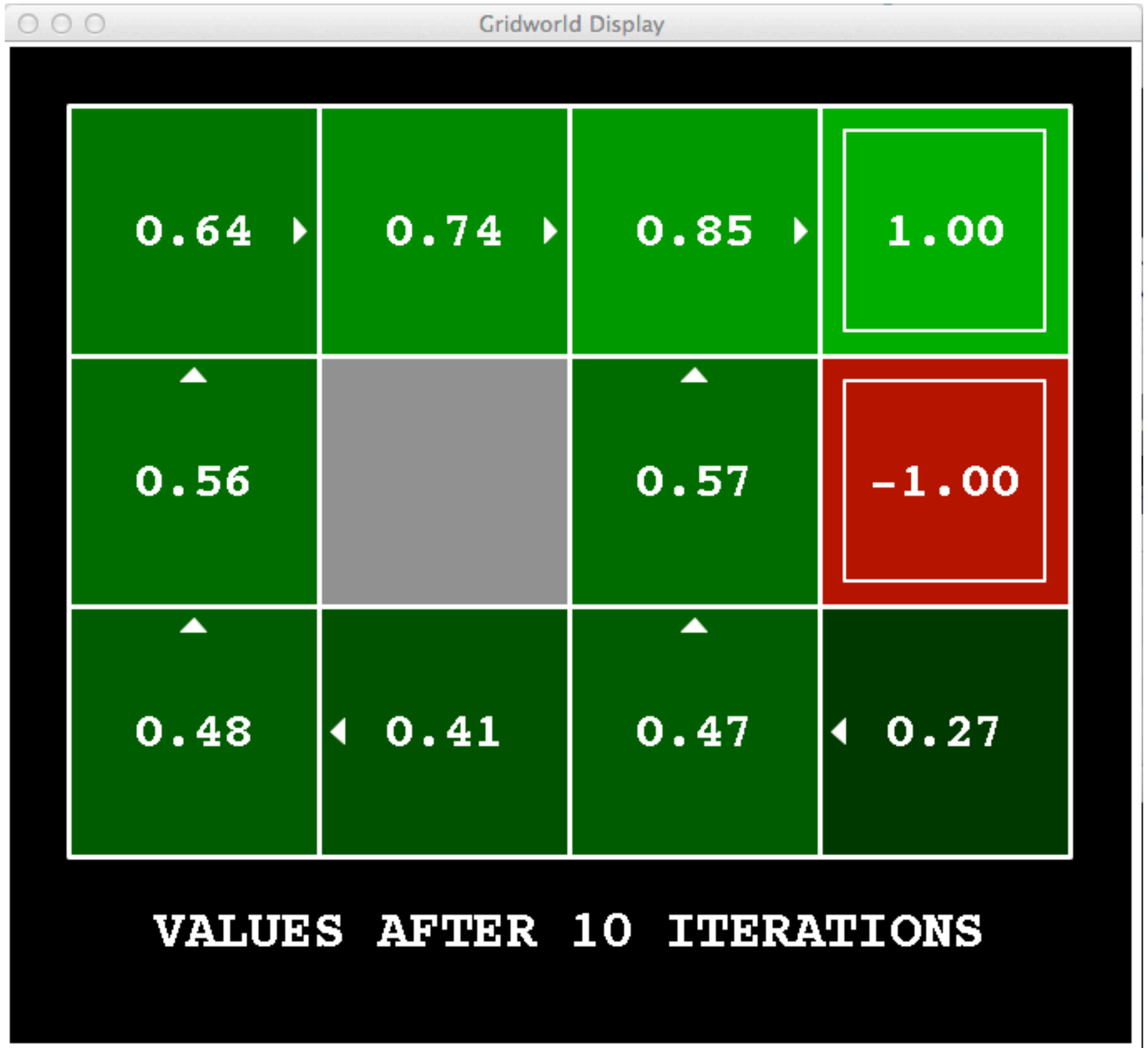
0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.55		▴ 0.57	-1.00
▴ 0.46	0.40 ▸	▴ 0.47	◂ 0.27

VALUES AFTER 9 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=10



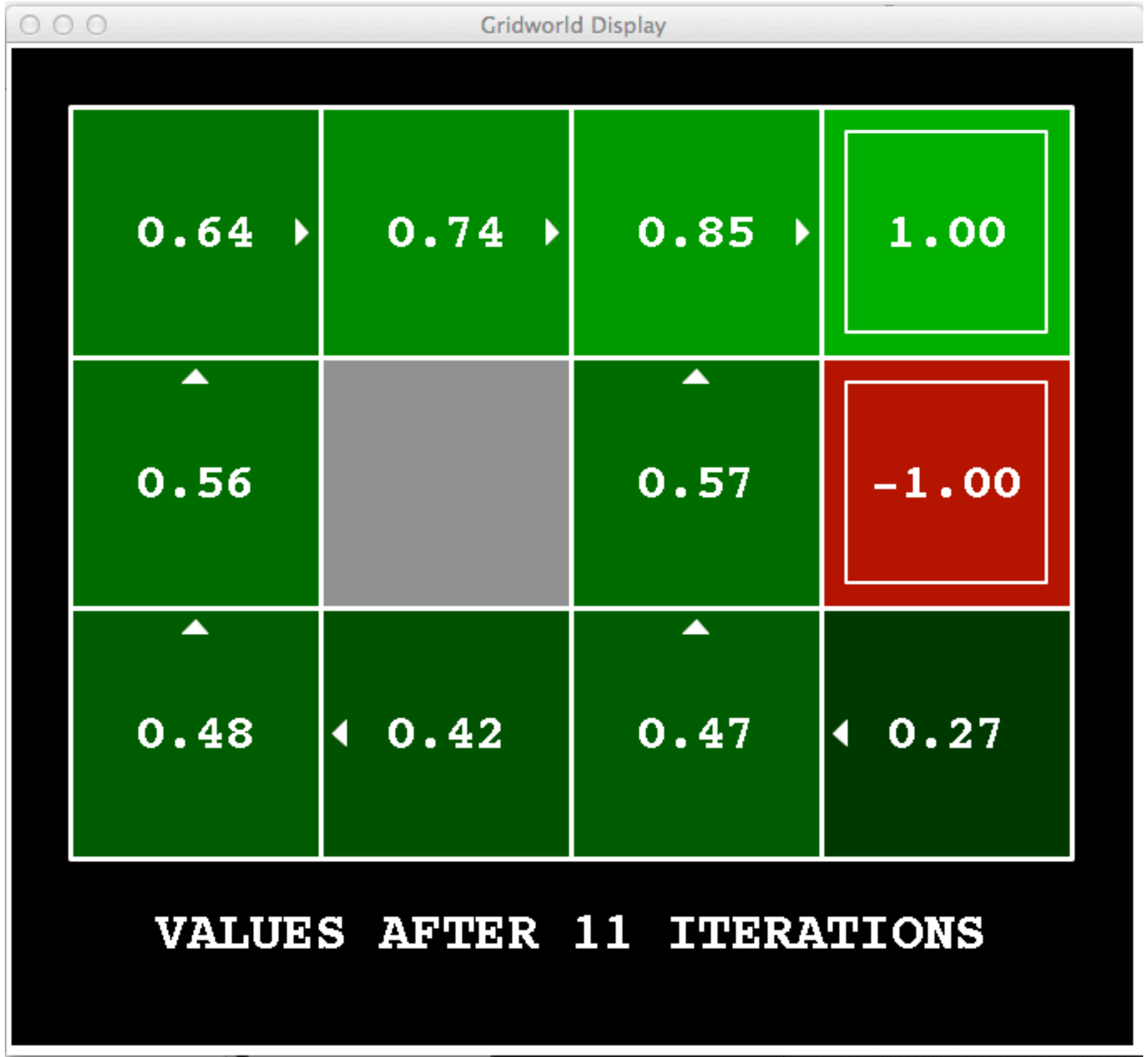
0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.56		▲ 0.57	-1.00
▲ 0.48	◀ 0.41	▲ 0.47	◀ 0.27

VALUES AFTER 10 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=11



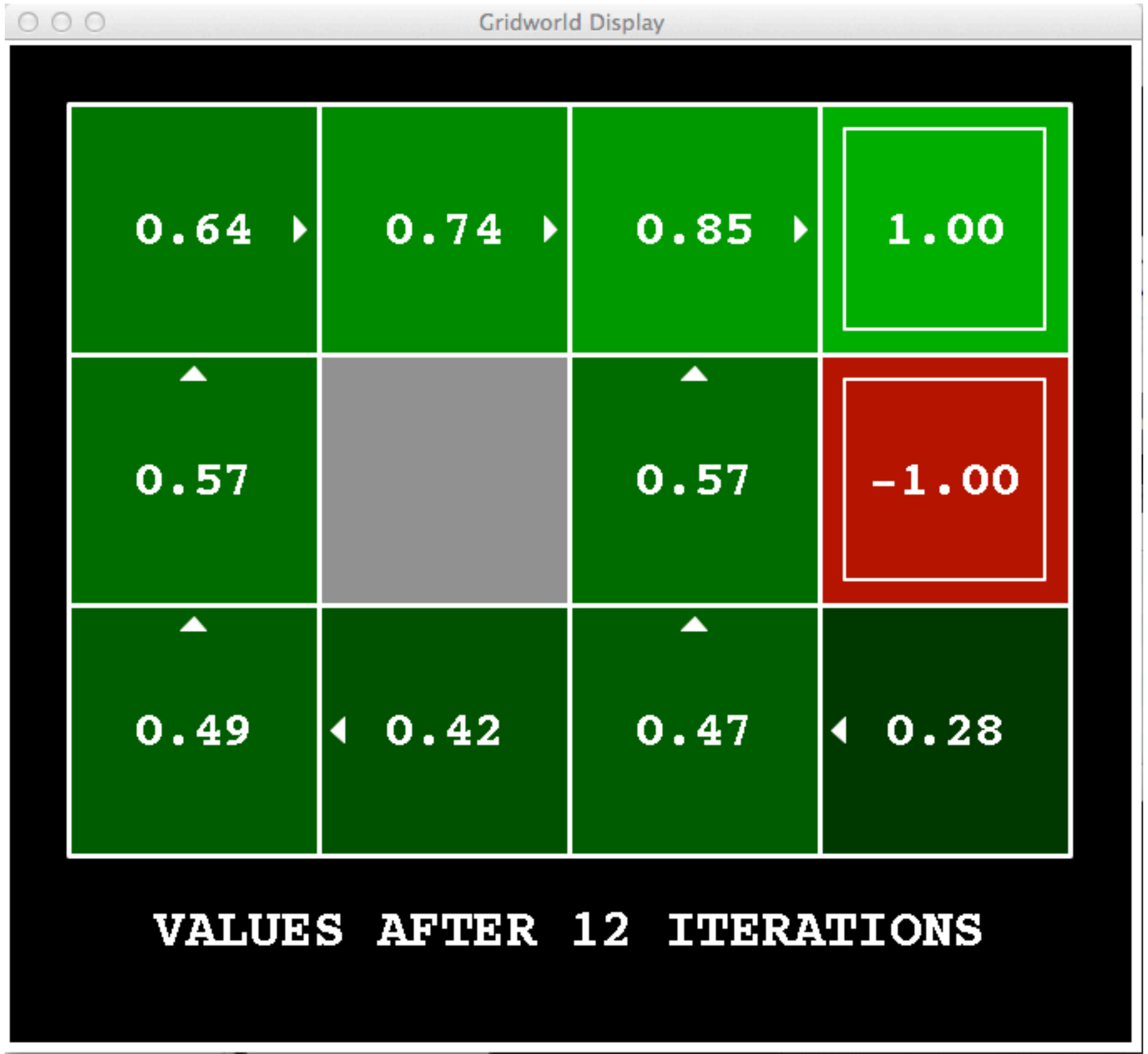
0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.56		▲ 0.57	-1.00
▲ 0.48	◀ 0.42	▲ 0.47	◀ 0.27

VALUES AFTER 11 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=12



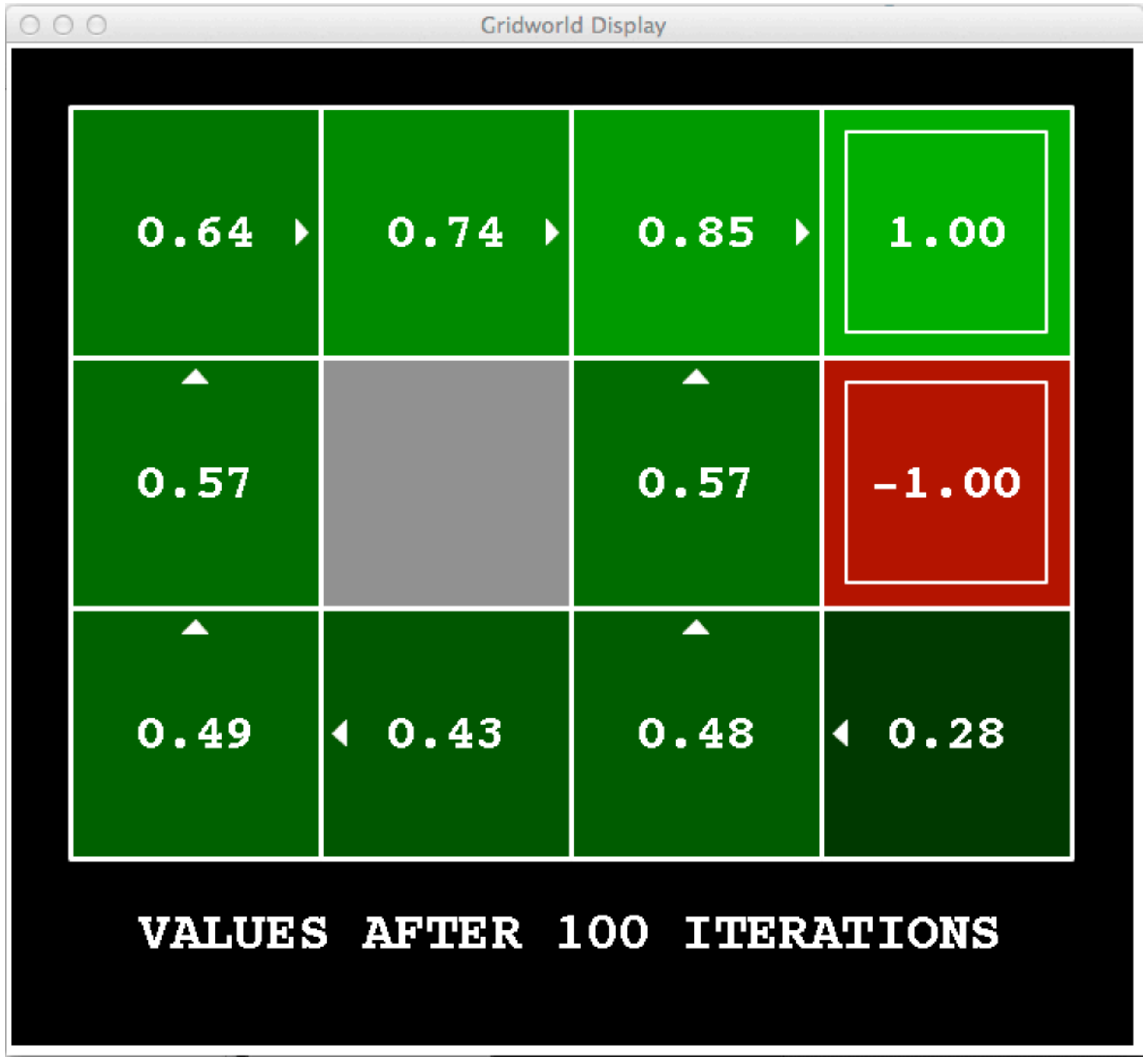
0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.57		▲ 0.57	-1.00
▲ 0.49	◀ 0.42	▲ 0.47	◀ 0.28

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=100



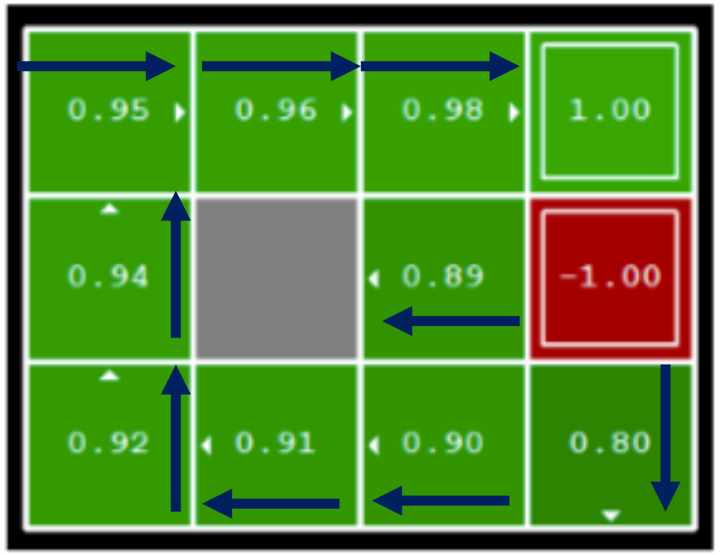
0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.57		▴ 0.57	-1.00
▴ 0.49	◂ 0.43	▴ 0.48	◂ 0.28

VALUES AFTER 100 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Extracting Policy from the Optimal Value Function

- **Our goal is to determine the policy for the MDP**
- **Step I: Estimate the optimal values V*(s)**
  - Through Value Iteration algorithm
- **Step II: Policy Extraction**
  - Obtain the policy implied by the values (using 1-step look ahead).
  - Use the policy to act in the environment.



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

Asynchronous Dynamic Programming

- Dynamic programming methods (Bellman updates) used synchronous updates (all states in parallel)
- Asynchronous DP
  - backs up states individually, in any order
  - can significantly reduce computation
  - guaranteed to converge if all states continue to be selected

In place Dynamic Programming

- Synchronous value iteration stores two copies of value function

$$\text{for all } s \text{ in } \mathcal{S}: \quad v_{\text{new}}(s) \leftarrow \max_a \mathbb{E}\left[R_{t+1} + \gamma v_{\text{old}}(S_{t+1}) \mid S_t = s\right]$$

$$v_{old} \leftarrow v_{new}$$

- In-place value iteration only stores one copy of value function

$$\text{for all } s \text{ in } \mathcal{S}: \quad v(s) \leftarrow \max_a \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]$$

Prioritized Sweeping

- Use magnitude of Bellman error to guide state selection, e.g.

$$\left| \max_a \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right] - v(s) \right|$$

- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue

Problems with Value Iteration

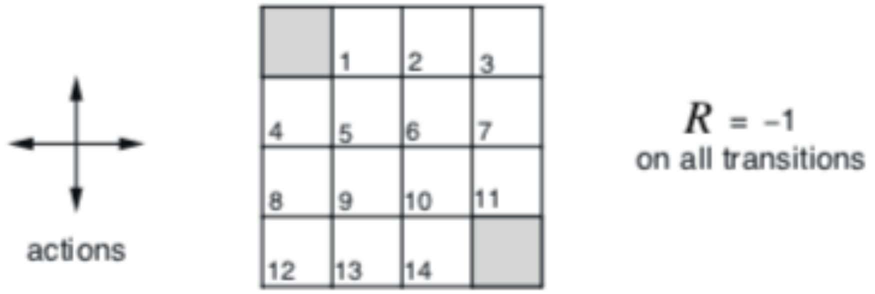
- **Value Iteration**
  - Repeats the bellman updates

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

- **Problems**
  - Slow: $O(S^2A)$ per iteration
  - The “max” at each state rarely changes. The policy often converges long before the values converge.

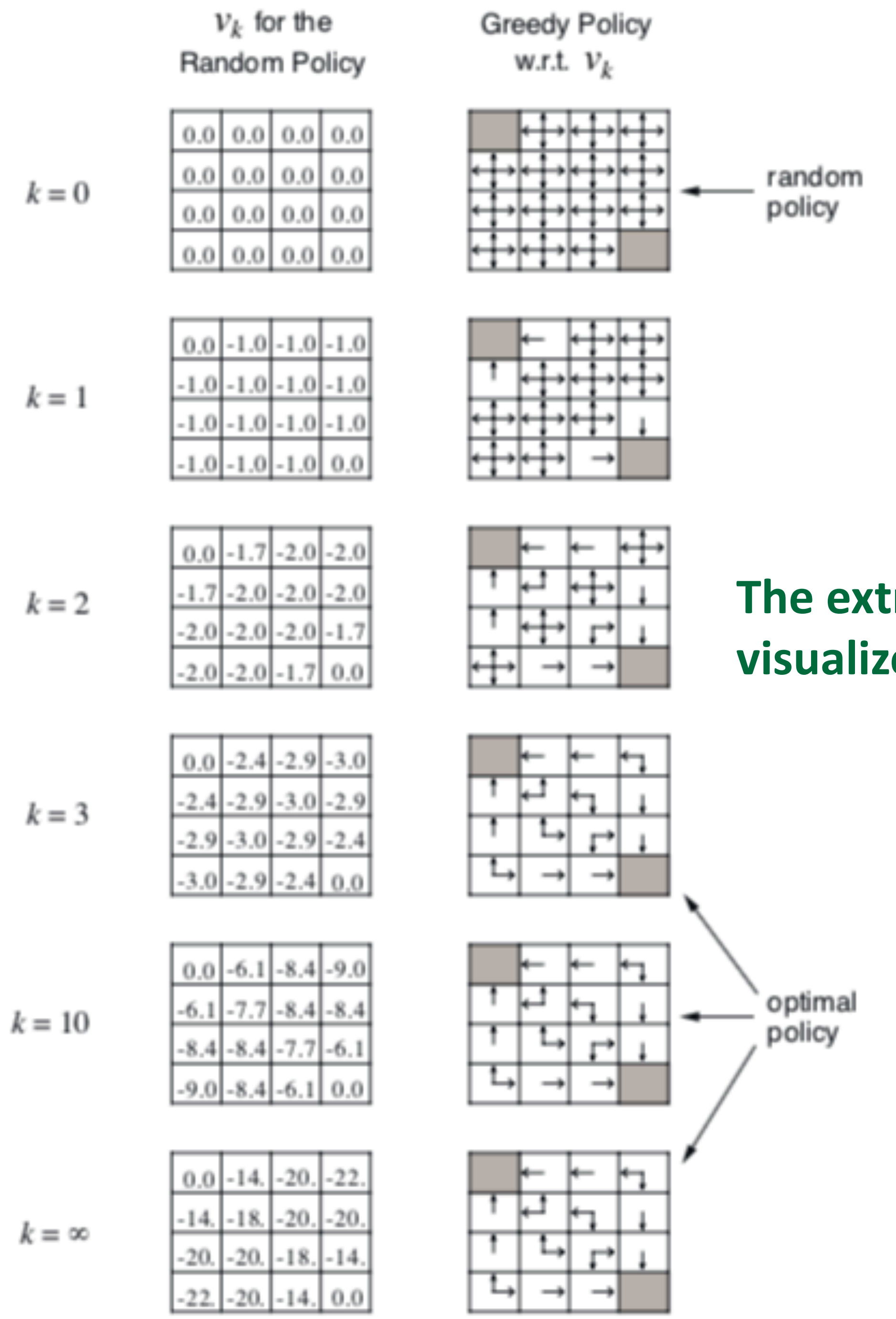
Example

Example: 4*4 Grid World MDP



The optimal policy converges before the values converge.

The optimal policy can be obtained even when the utility function estimate is inaccurate.



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

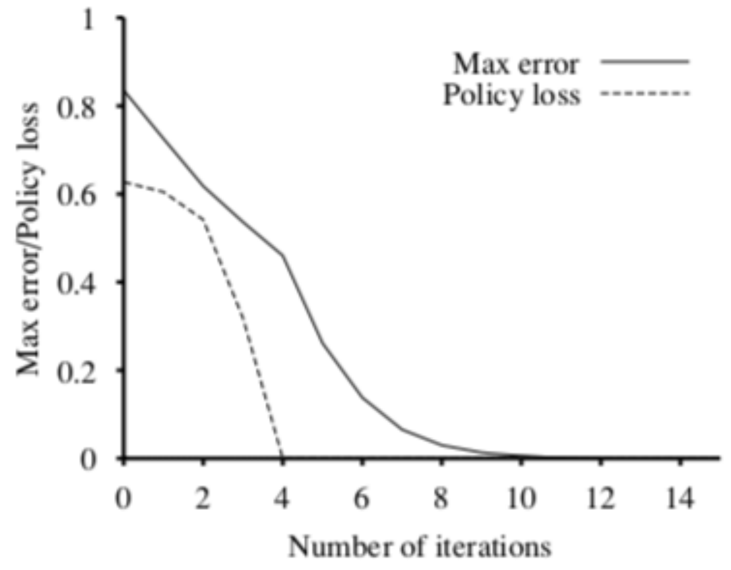
$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

The extracted policy visualized at each stage.

Policy Iteration

- **Observations**
  - It is possible to get an optimal policy even when the utility function is inaccurate.
  - If one action is clearly better than all others, then the exact magnitude of the utilities on states involved need not be precise.
- Policy iteration provides an alternative way to arrive at optimal policies.



The maximum error of the utility estimates and the policy loss as a function of the number of iterations of value iteration.

Policy Iteration

- **Policy Iteration:**
  - **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.

- Converges to the optimal policy. In practice, can converge faster*.

*conditions beyond the scope of this course.

Step I: Policy Evaluation

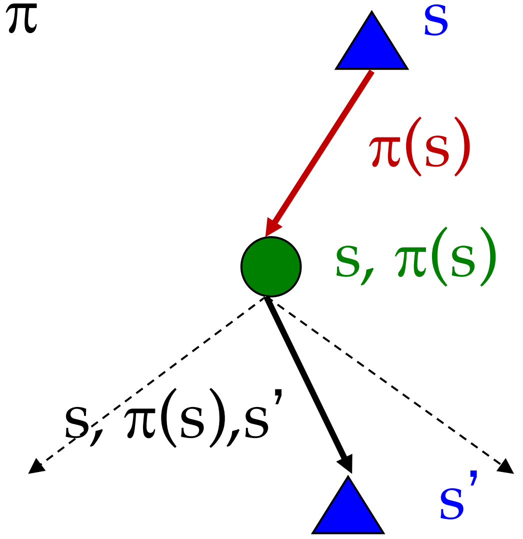
Evaluate the utilities for states under the actions provided by the given policy

- Policy Evaluation: Calculating the value function V for a fixed policy π

- Perform a number of simplified Bellman updates (given a policy)

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

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



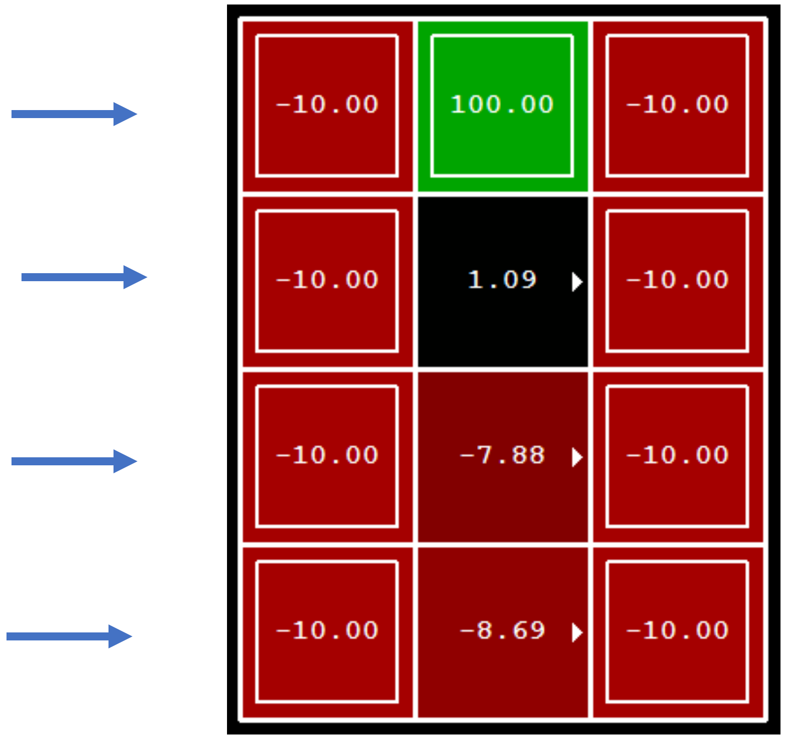
- The Bellman equations are a linear system of equations without the max operator.
  - Can be solved in $O(n^3)$ time by standard linear algebra solvers.
  - It is not necessary to obtain the exact solution.
  - Can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a good approximation of the utility values of the states.

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

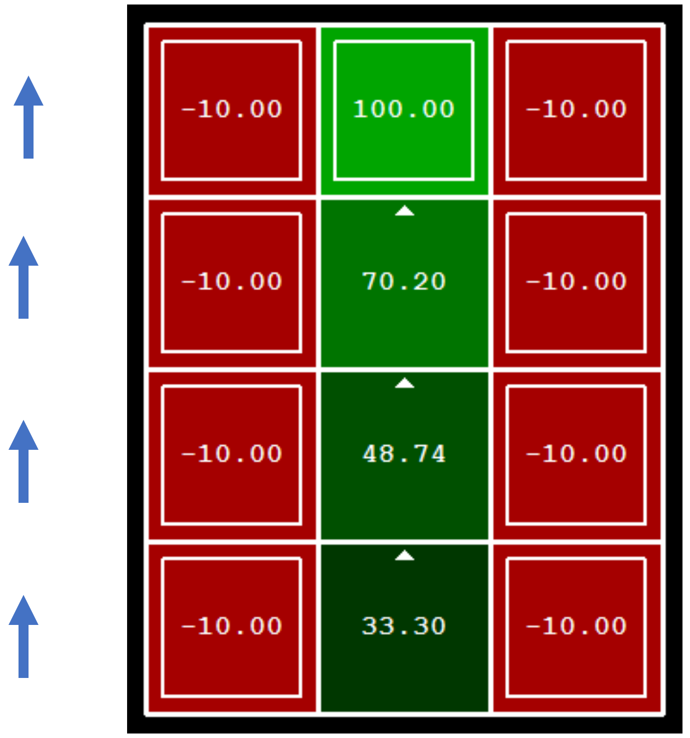
Toy Example: Evaluating two different policies

Bridge passing task. The agent must move towards the end of a bridge. Undesirable states to the left and the right.

Policy I: Always Go Right



Policy II: Always Go Forward

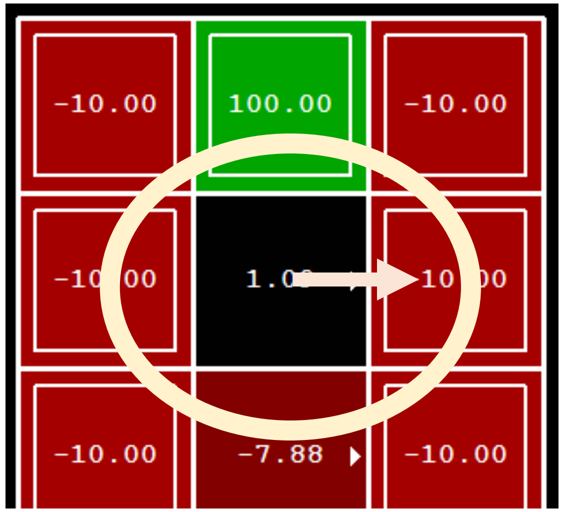


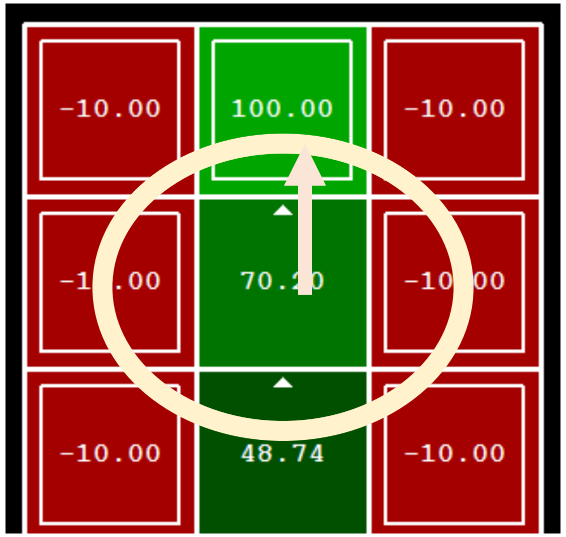
Step II: Policy Improvement

- Improvement
  - Given fixed utility values for states (obtained via policy evaluation)
  - Examine if there is a better policy (policy extraction) using one-step look-ahead

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

Toy example





Policy Iteration: Evaluation and Improvement

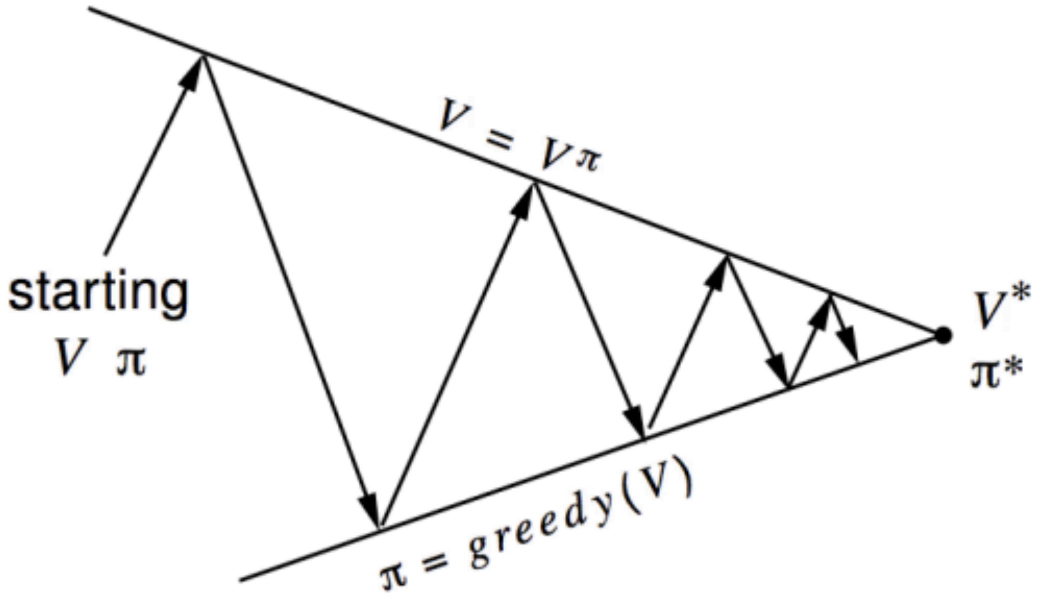
- **Pick an arbitrary policy**
- **Iterate (until the policy changes)**
  - **Policy Evaluation:** For fixed current policy π, find values with policy evaluation:
    - Iterate until values converge:

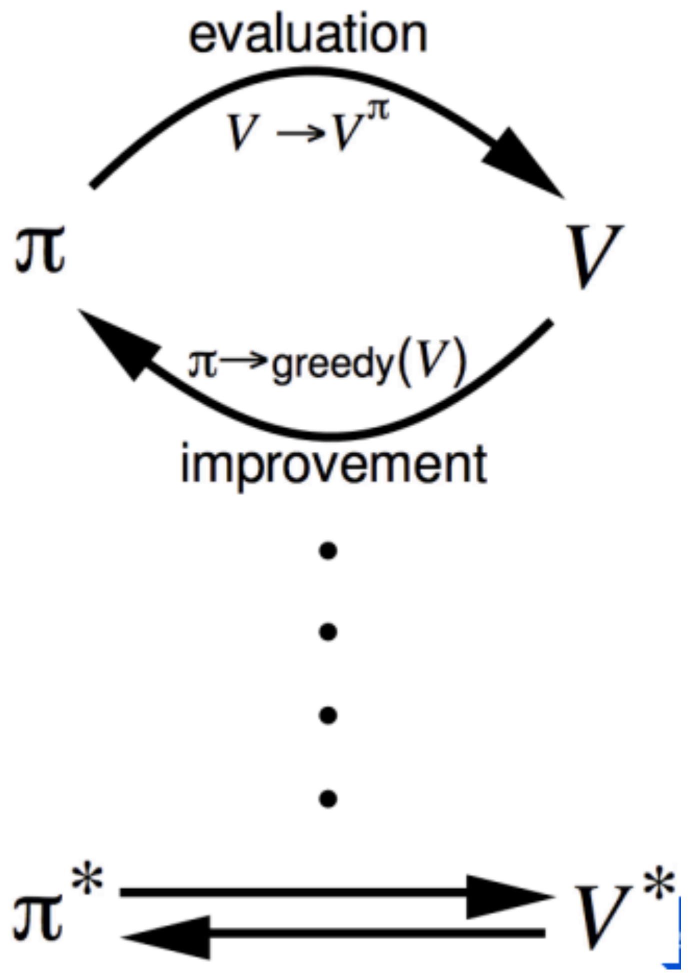
$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma\, V_k^{\pi_i}(s')\right]$$

  - **Policy Improvement:** For fixed values, get a better policy using policy extraction
    - One-step look-ahead:

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

Policy Iteration: Evaluation and Improvement

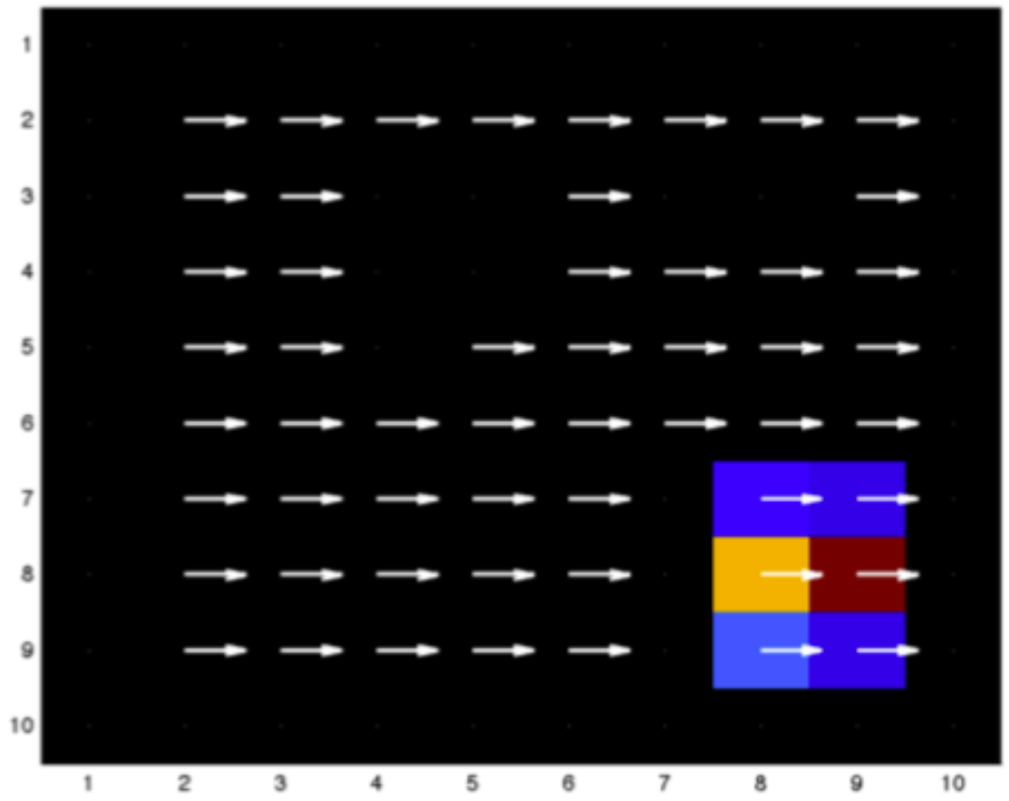


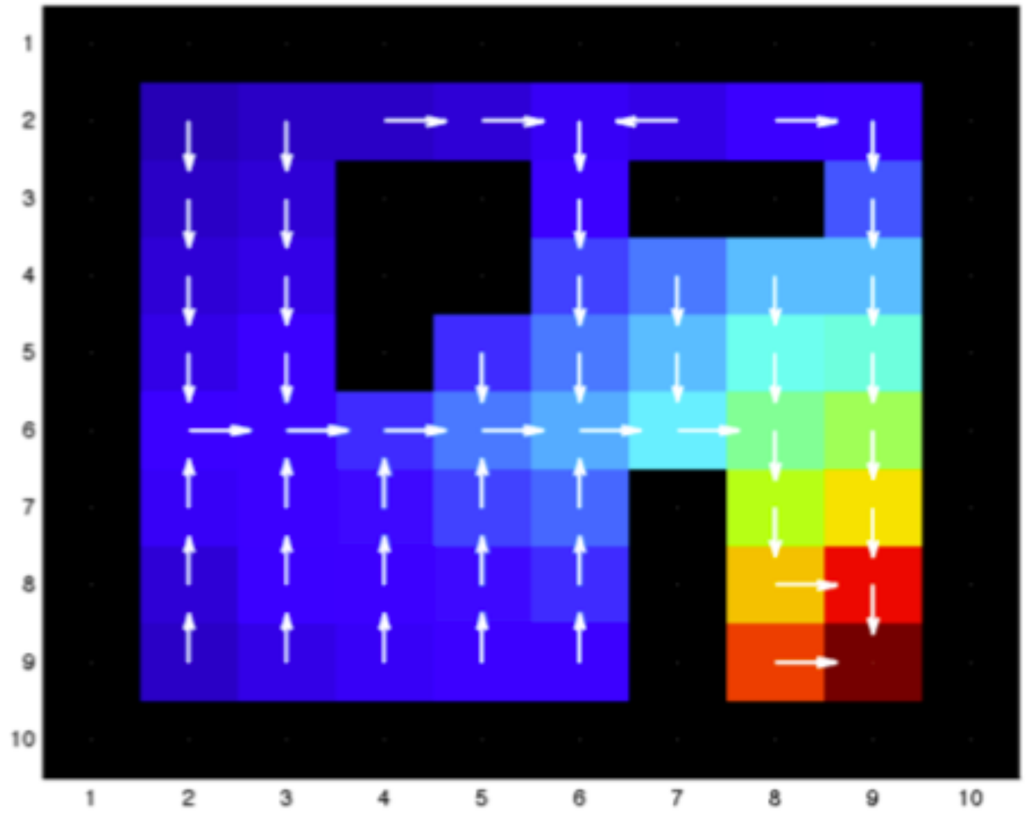


Policy evaluation Estimate v^π

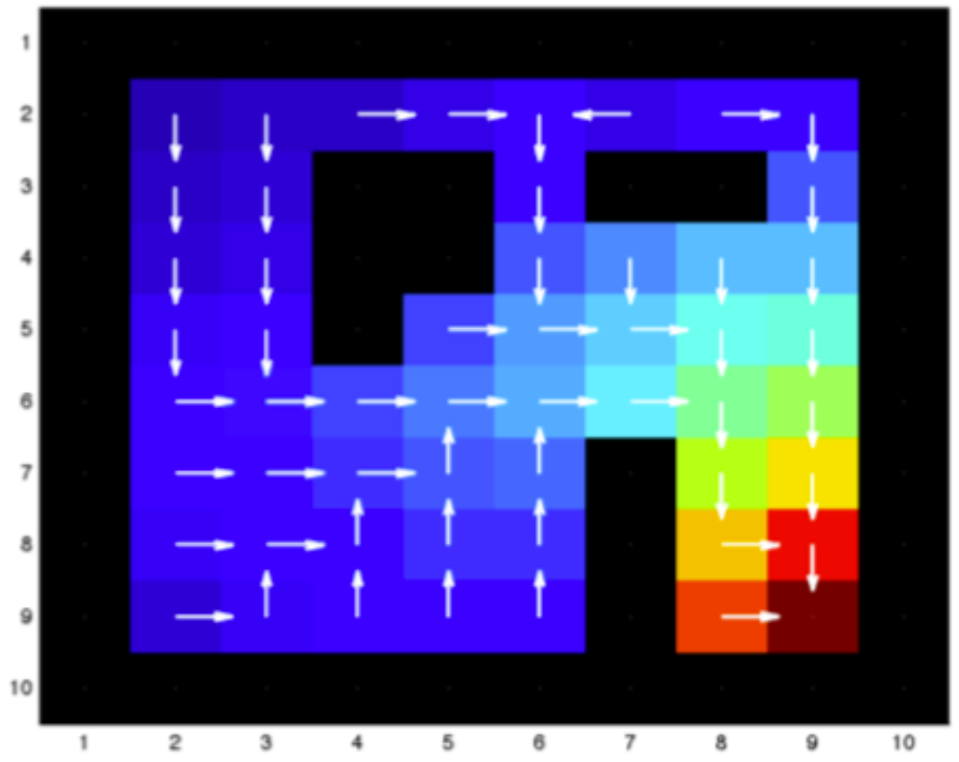
Policy improvement Generate $\pi' \geq \pi$

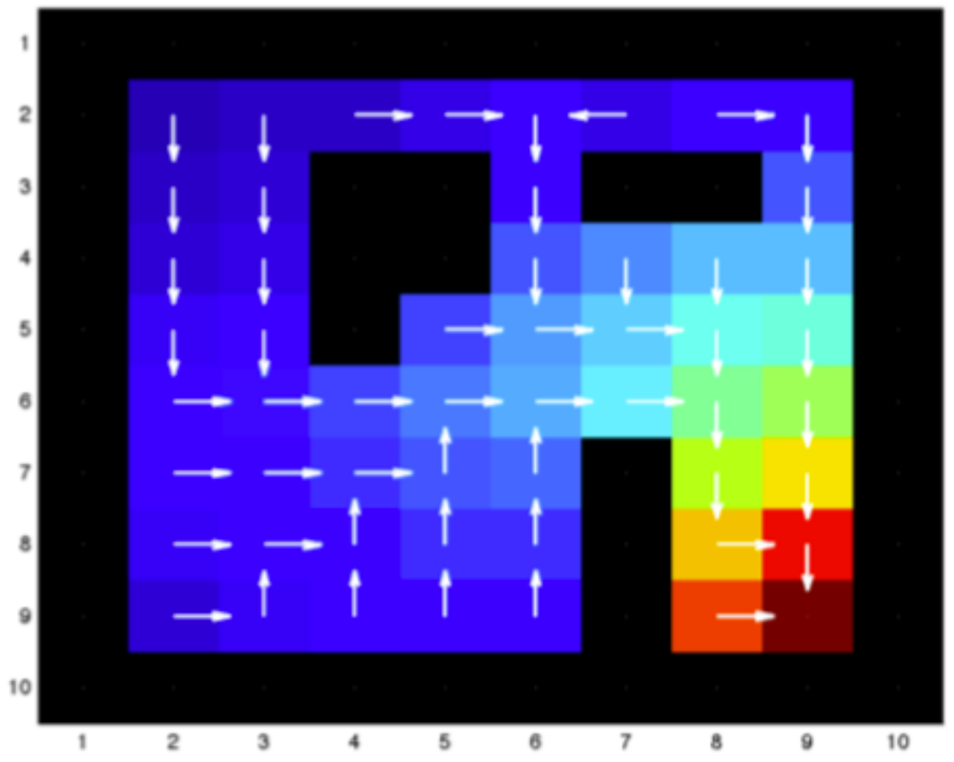
Policy Iteration: Example





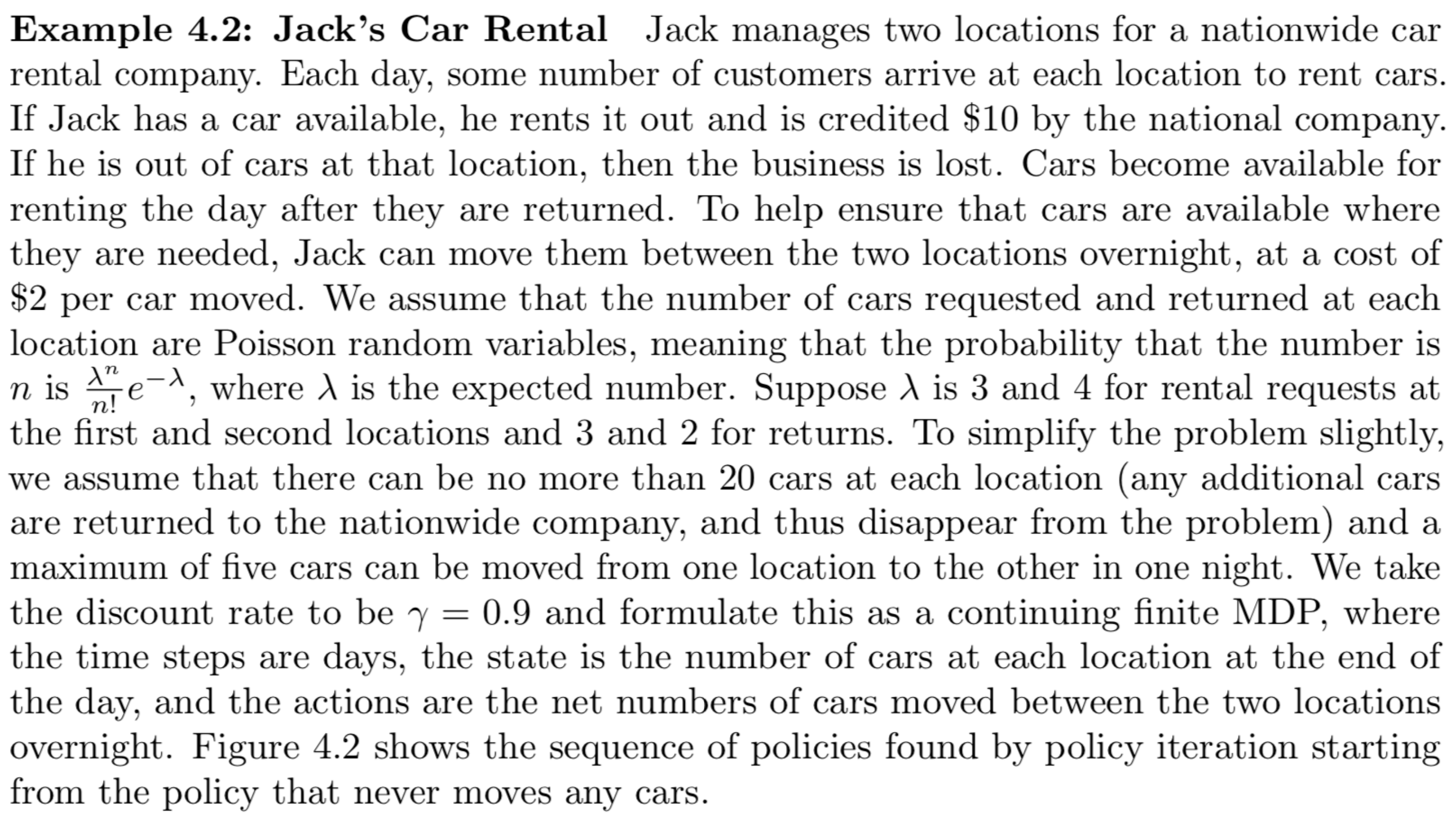
Policy Iteration: Example





Convergence in 4 iterations

Policy Iteration: Jack’s Car Rental



Example 4.2: Jack’s Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!}e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 4.2 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

Source: Sutton and Barto Section 4.2

Policy Iteration: Jack's Car Rental

- States: Two locations, maximum of 20 cars at each
- Actions: Move up to 5 cars overnight (-\$2 each)
- Reward: \$10 for each available car rented, $\gamma = 0.9$
- Transitions: Cars returned and requested randomly
  - Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!}e^{-\lambda}$
  - 1st location: average requests $= 3$, average returns $= 3$
  - 2nd location: average requests $= 4$, average returns $= 2$

Policy Iteration: Jack's Car Rental

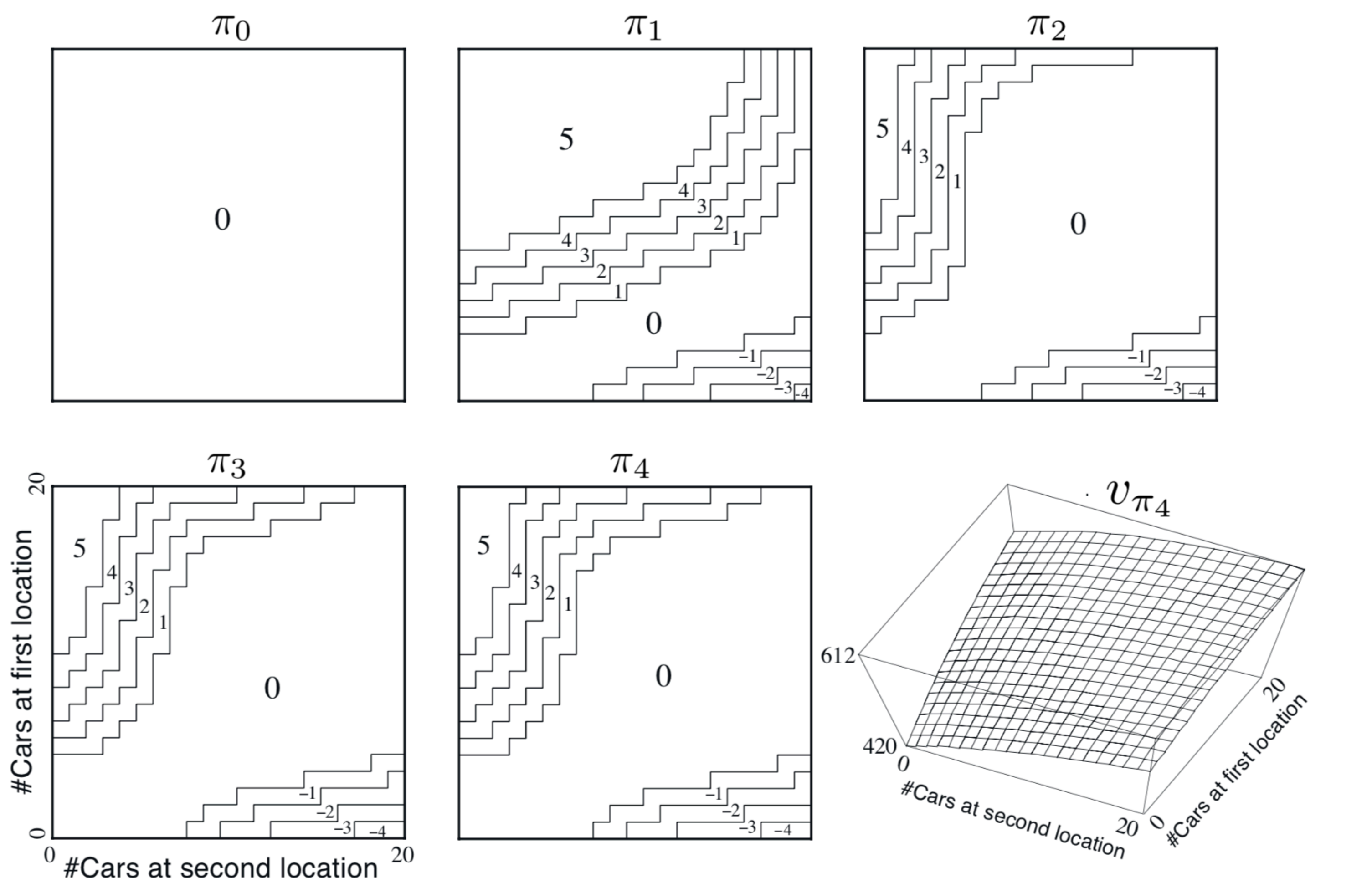


Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal. ■

Value Iteration and Policy Iteration

- **Both** value iteration and policy iteration compute the optimal values.
- Policy iteration has the advantage of **finite-time convergence** to the optimal policy.
- Both are **dynamic programs** for solving MDPs