Artificial Intelligence

Lecture 13：Markov Decision Processes

Xiaojin Gong

2022-05-23

Outline

- Markov Decision Processes
    - Problem Formulation
    - Value Iteration
    - Policy Iteration

Review

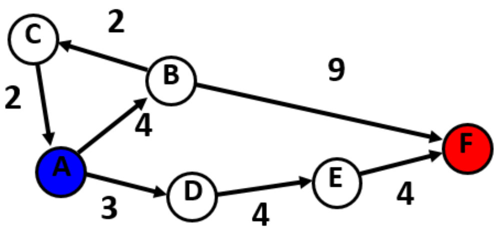
Problems solved by searching + Probabilistic graphical models

- Problem formulation
- Solving by searching
  - Deterministic environment

- Representation
- Inference
- Learning

Uncertainty

Planning in deterministic environments → Planning in stochastic environments



state s, action a

deterministic

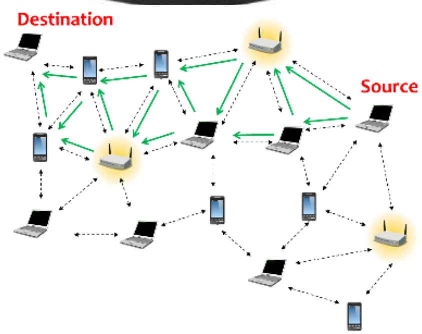
state Successor(s, a)

state s, action a

random

state s_1 state s_2 state s_3

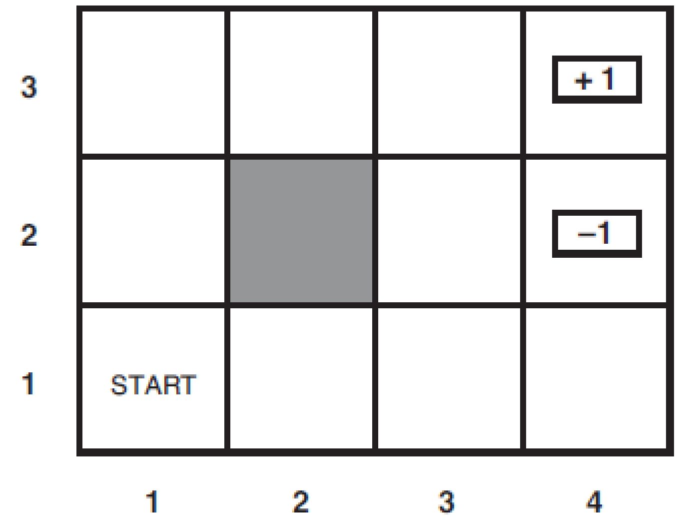
Applications



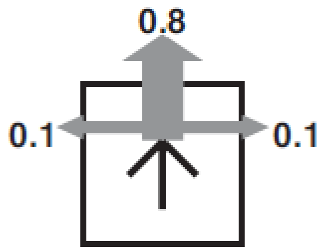
- Cleaning robot: decide where to move, but actuators may fail, hit obstacles, etc.
- Network routing: decide which server to go, but server may crash, etc.
- Agriculture: decide what to plant, but don't know weather and thus crop yield
- Elevator scheduling
- Aircraft navigation
- Resource allocation

Example: Robot Navigation

- The problem:
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement:
  - 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



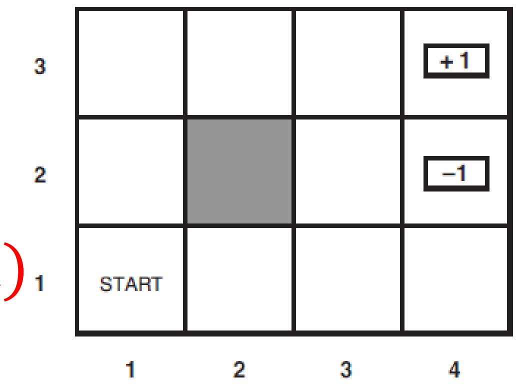
- The agent receives rewards each time step
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)



- Goal: maximize the utility (sum of rewards)

Markov Decision Processes

- An Markov Decision Process (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 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 $R(s)$
  - A start state
  - Maybe a terminal state
- Solution:
  - A policy π gives an action for each state
- MDPs are non-deterministic search problems



Markov Decision Processes

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

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

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

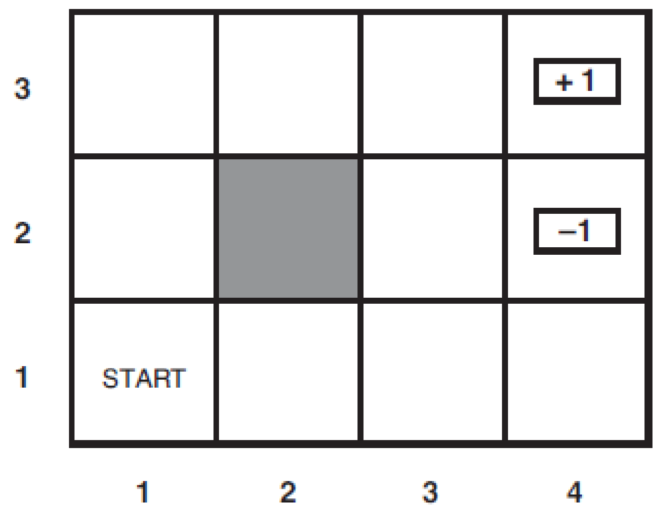
- This is just like search, where the successor function could only depend on the current state (not the history)

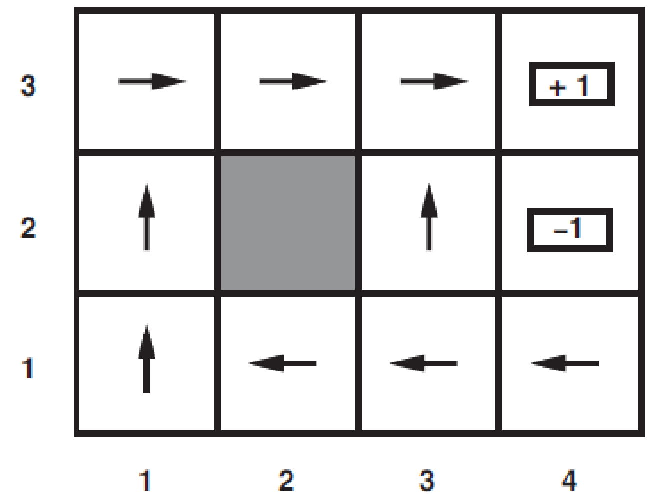
Markov Decision Processes

- A sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a Markov Decision Process.

Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

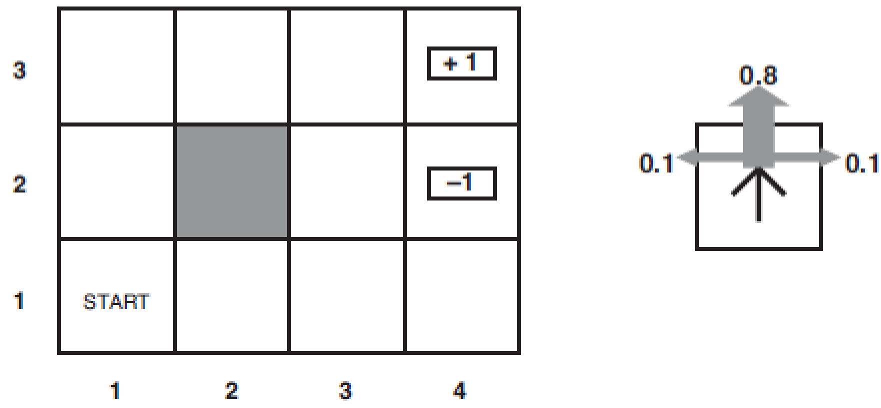


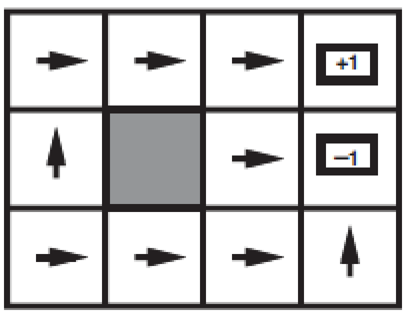


$R(s) = -0.04$

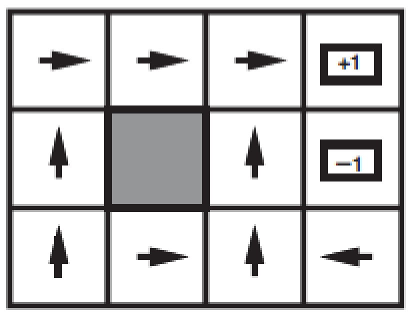
Optimal Policies

- Balancing of risk and reward

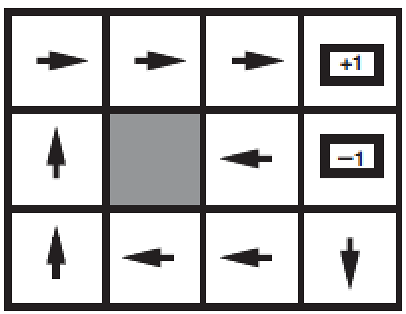




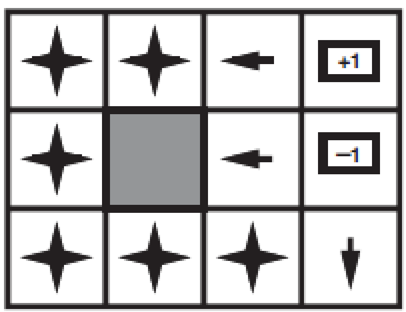
$R(s) < -1.6284$



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



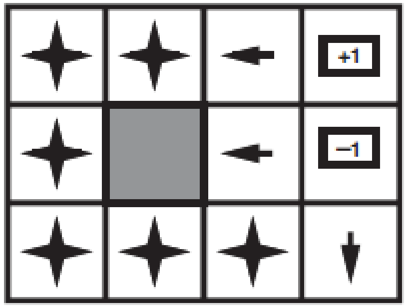
$-0.0221 < R(s) < 0$



$R(s) > 0$

Utilities of Sequences

- 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)



$R(s) > 0$

- Infinite horizon:
  - No fixed time limit
  - The optimal policy is stationary

Utilities of Sequences

- Theorem: if we assume stationary preferences:

$$[s_1, s_2, \ldots] \succ [s'_1, s'_2, \ldots]$$
$$\Updownarrow$$
$$[s_0, s_1, s_2, \ldots] \succ [s_0, s'_1, s'_2, \ldots]$$

- Then: there are only two ways to define utilities

  - Additive rewards:

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots \ .$$

  - Discounted rewards:

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

Utilities of Sequences

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

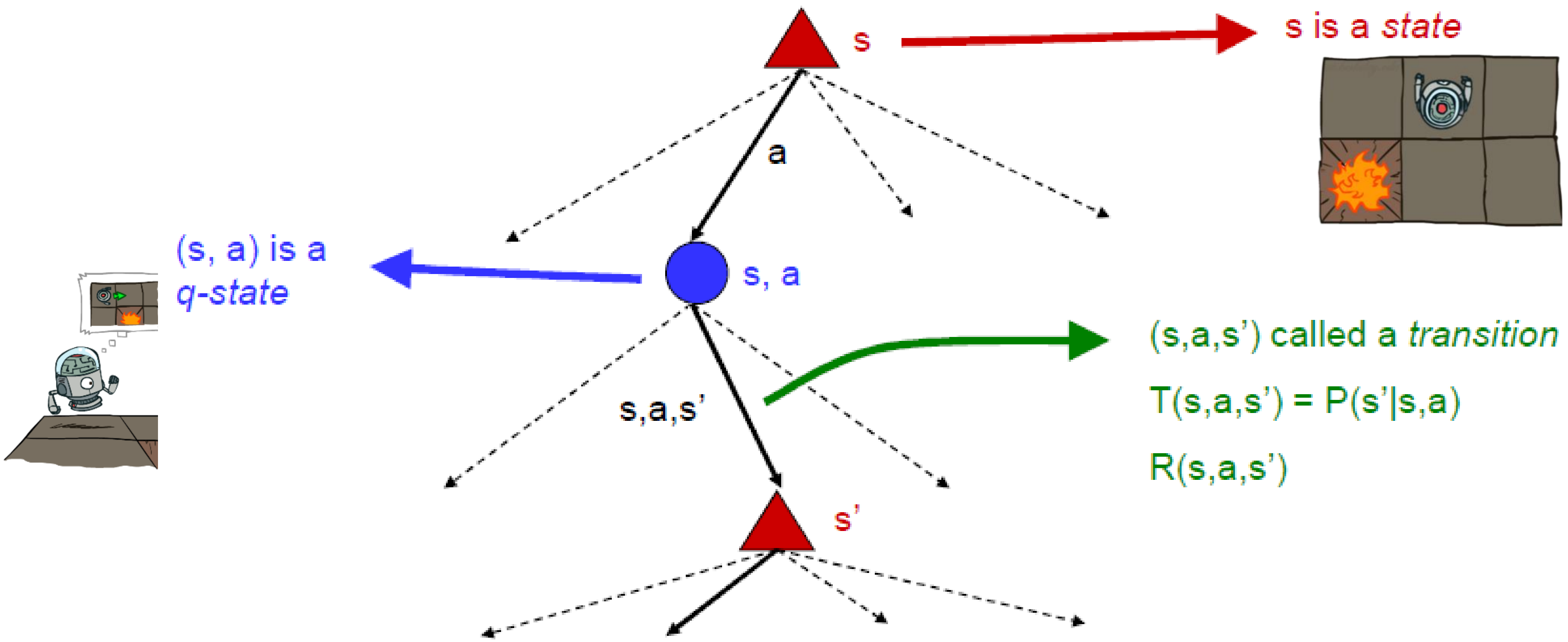
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge

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

$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1-\gamma)$$

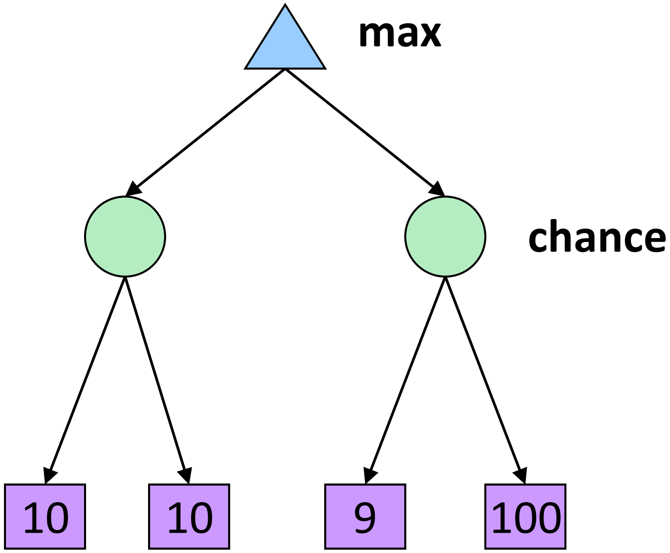
MDP Search Trees

- Each MDP state gives an expectimax-like search tree



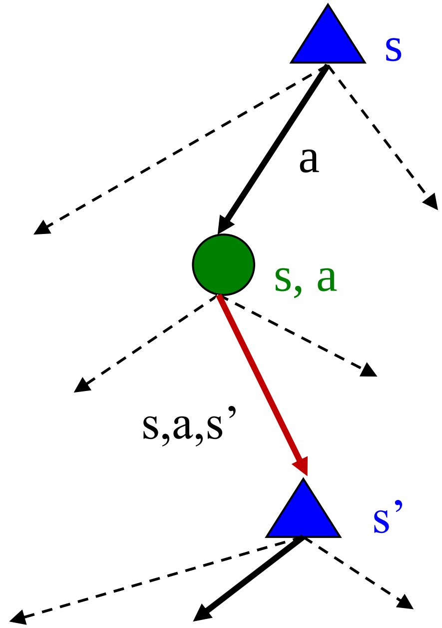
Expectimax Search Tree

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities, i.e. take weighted average (expectation) of children



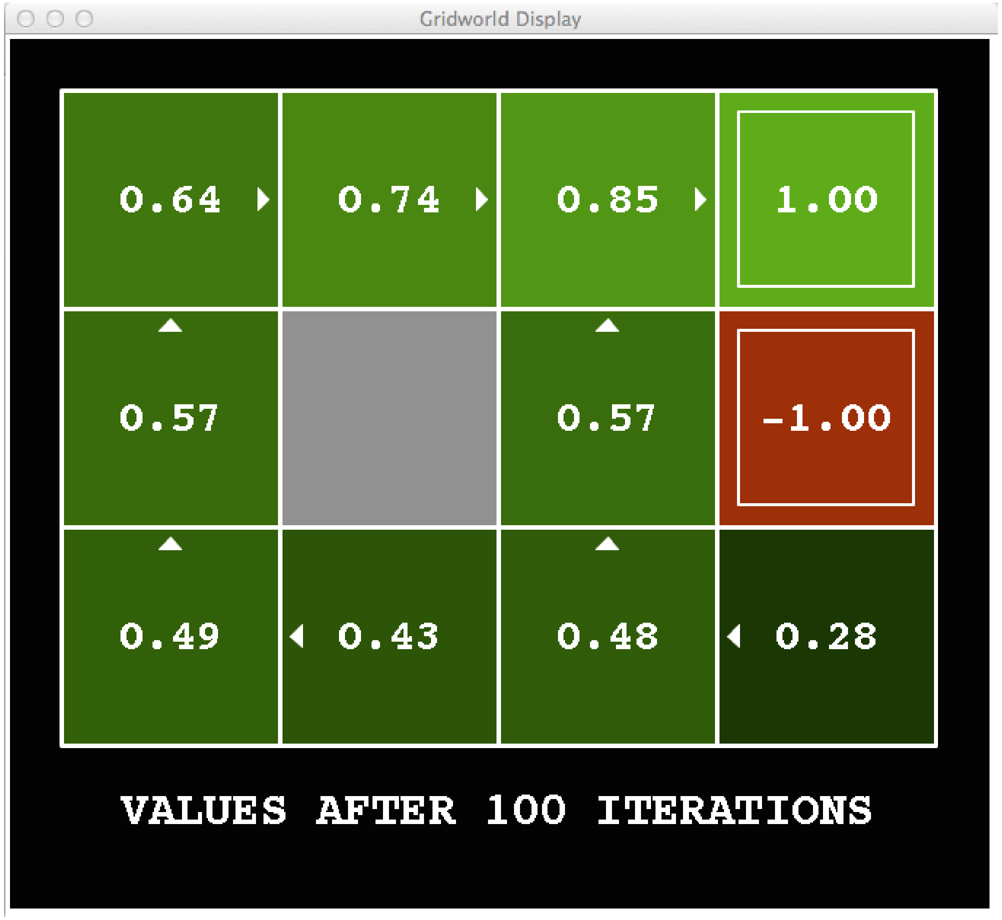
Optimal Quantities

- 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:
  - π*(s) = optimal action from state s
- Optimal values define optimal policies!



Optimal Quantities

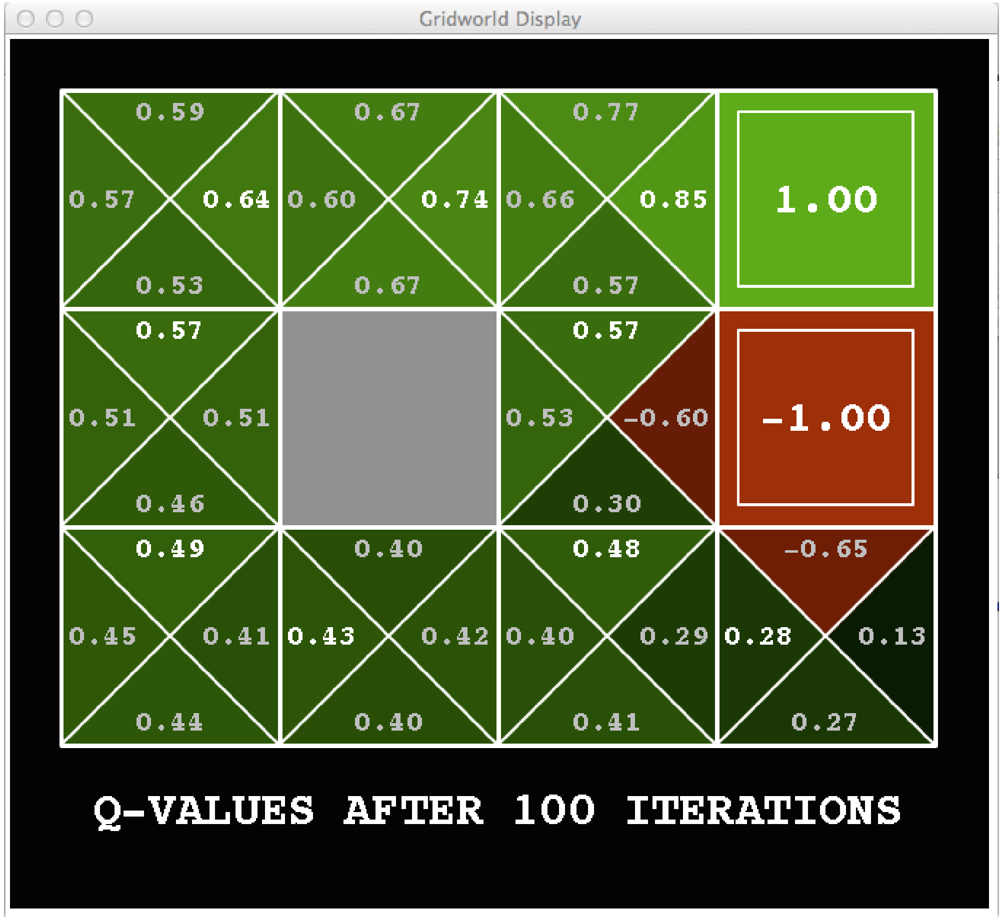
- Example: Gridworld V Values



Noise = 0.2
Discount = 0.9
Living reward = 0

Optimal Quantities

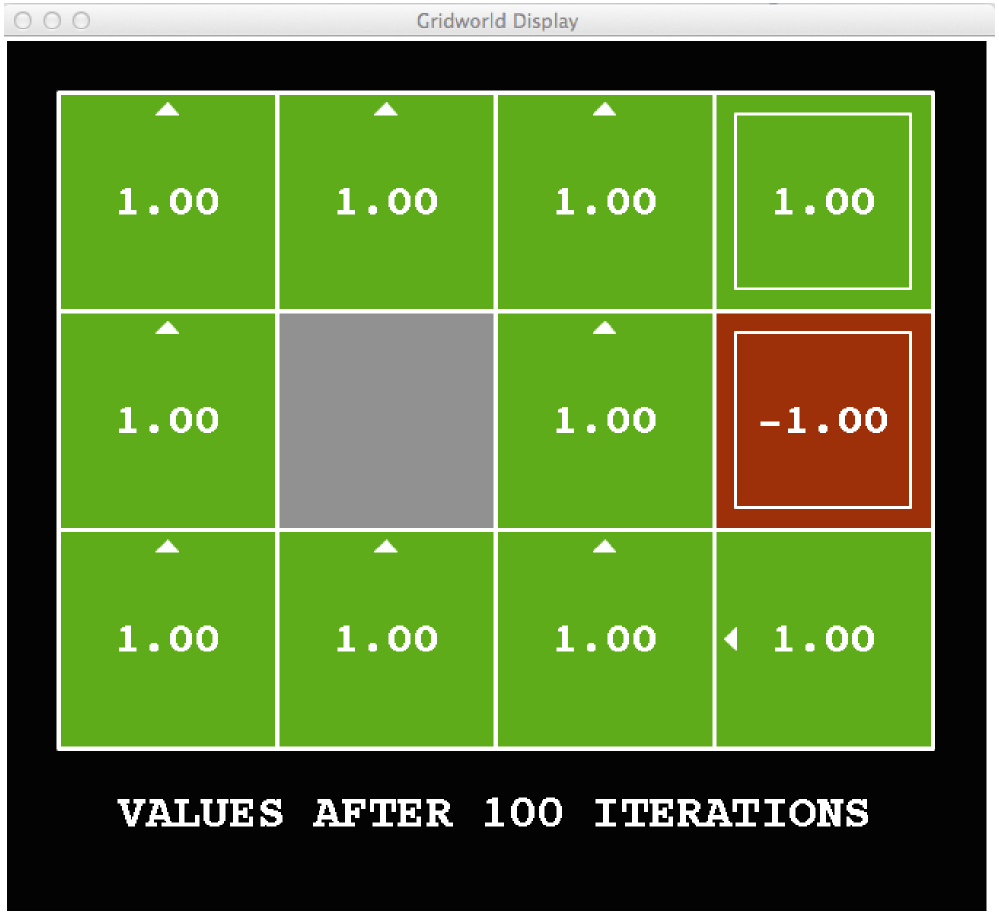
- Example: Gridworld Q Values



Noise = 0.2
Discount = 0.9
Living reward = 0

Optimal Quantities

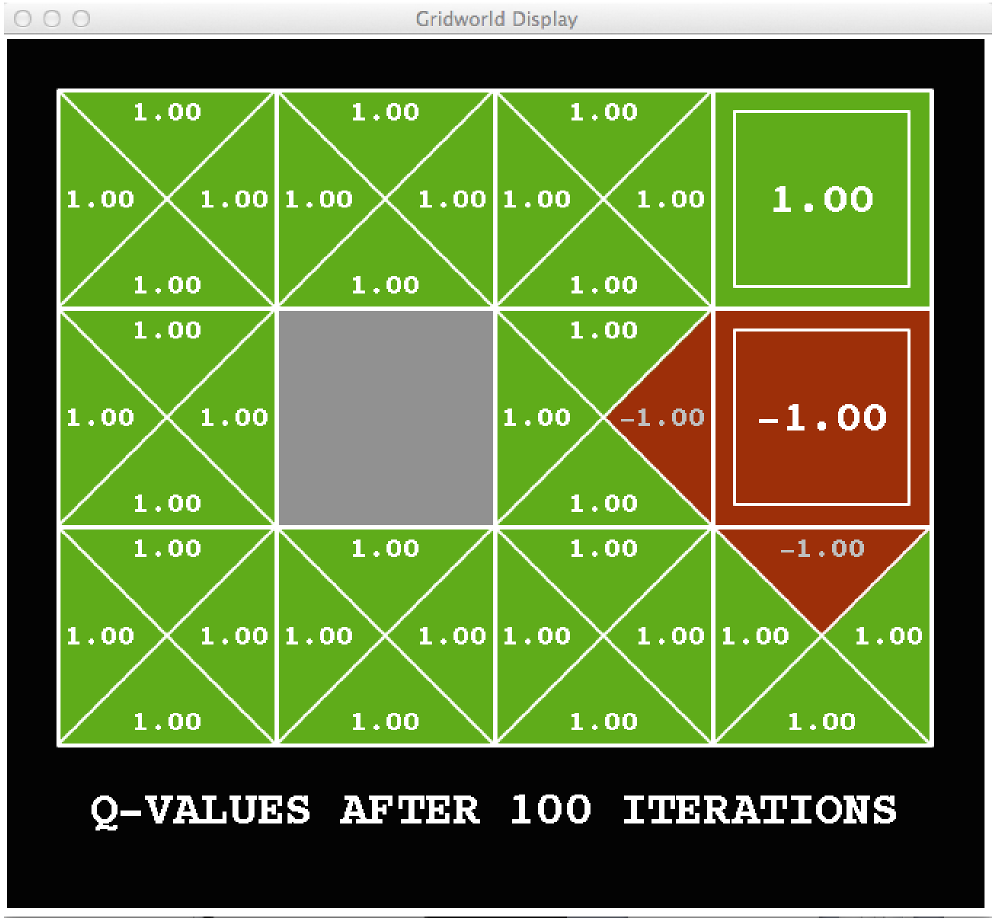
- Example: Gridworld V Values



Noise = 0
Discount = 1
Living reward = 0

Optimal Quantities

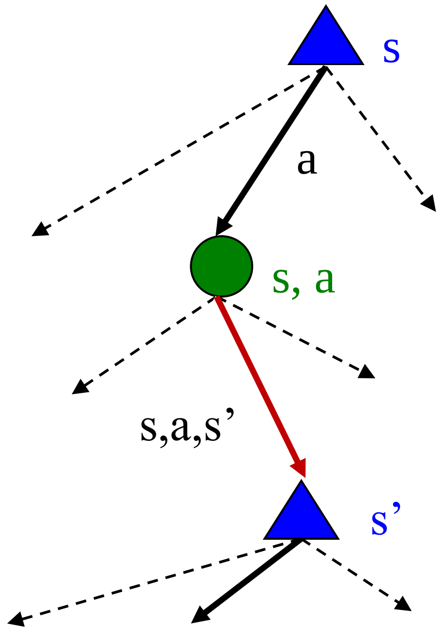
- Example: Gridworld Q Values



Noise = 0
Discount = 1
Living reward = 0

Recap: Defining MDPs

- Markov decision processes:
  - States S
  - Actions A
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount γ)
  - Start state s_0



s is a *state*

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

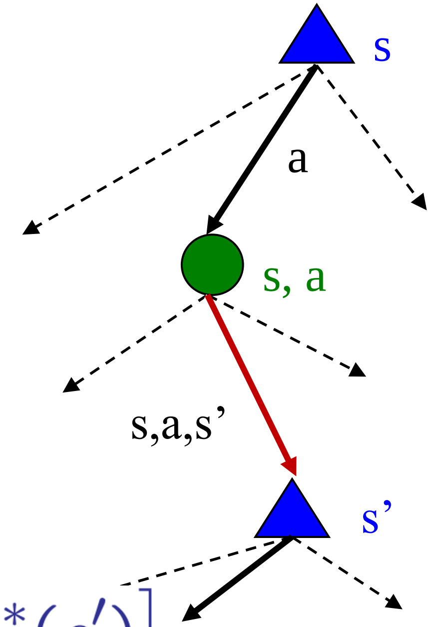
(s,a,s') is a *transition*

- Quantities:
  - Policy = map of states to actions
  - Utility = sum of discounted rewards
  - Values = expected future utility from a state (max node)
  - Q-Values = expected future utility from a q-state (chance node)

Values of States

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

- Recursive definition of value:



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

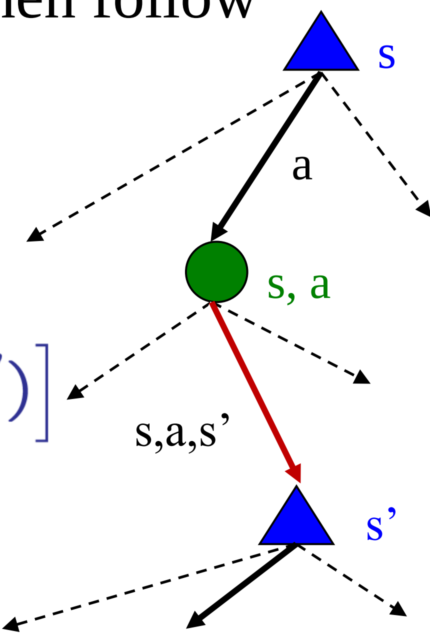
$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

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

The Bellman Equation

$$\boxed{U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')}$$

- Optimal rewards = maximize over first action and then follow optimal policy

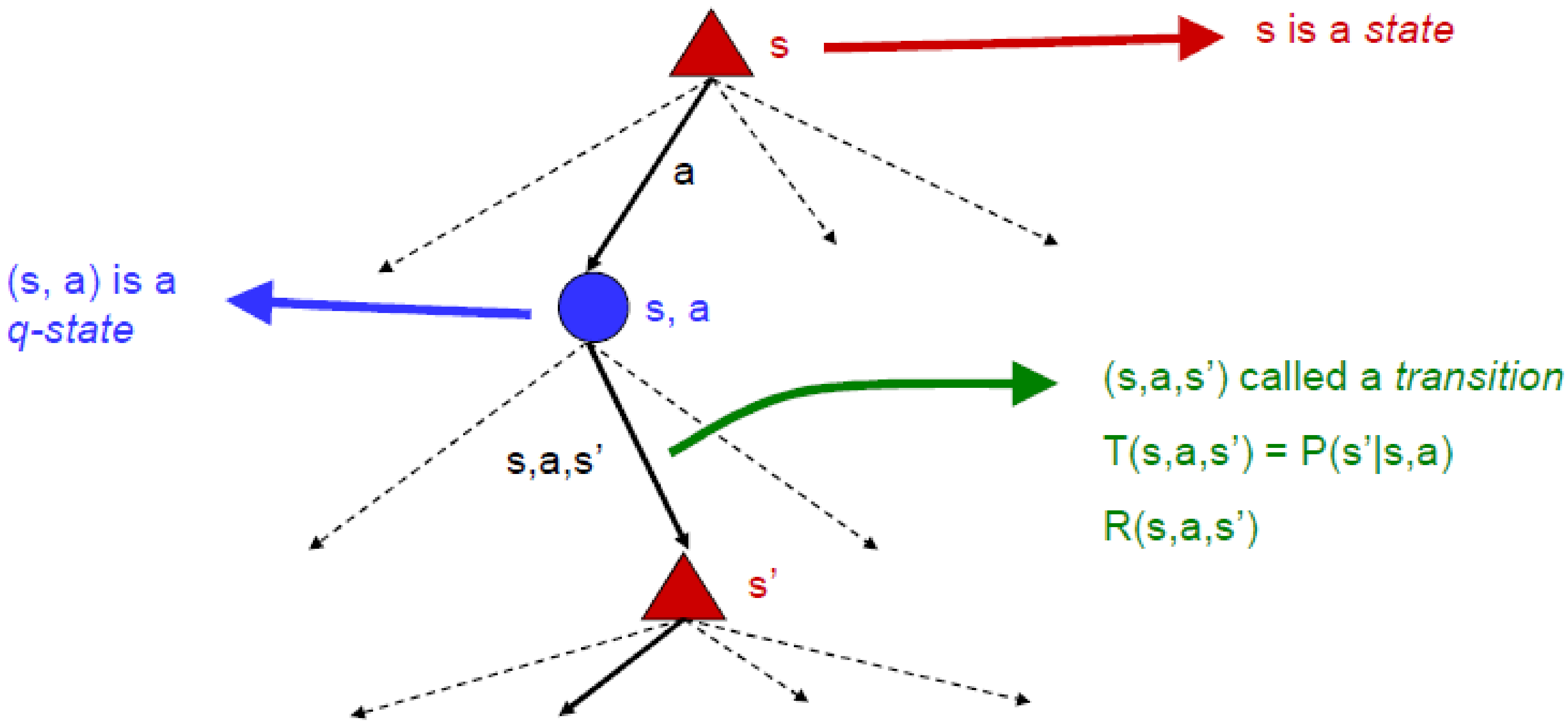


- Recursive definition of value:

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

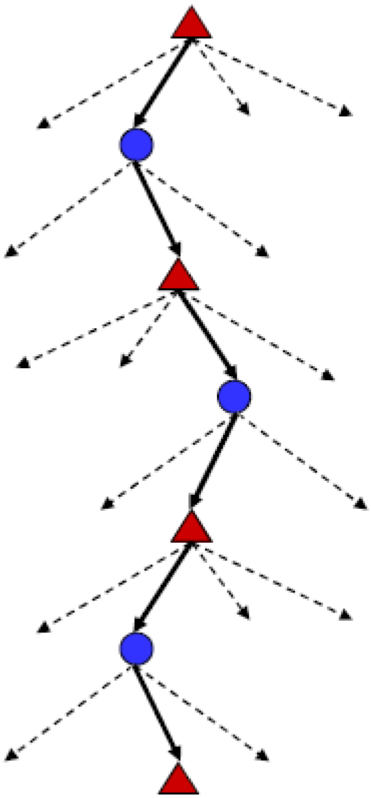
MDP Search Trees

- Each MDP state gives an expectimax-like search tree



MDP Search Trees

- Problems:
  - This tree is usually infinite
  - Same states appear over and over
  - We would search once per state
- Idea: Value iteration
  - Compute optimal values for all states all at once using successive approximations
  - Will be a bottom-up dynamic program similar in cost to memoization
  - Do all planning offline, no replanning needed!



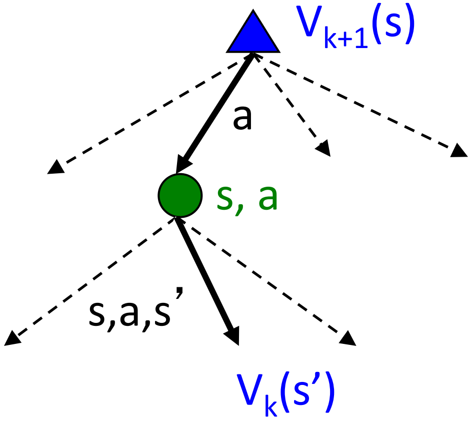
Value Iteration

- 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 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

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



The Value Iteration Algorithm

```
function VALUE-ITERATION(mdp, ϵ) returns a utility function
   inputs: mdp, an MDP with states S, actions A(s), transition model P(s′ | s, a),
              rewards R(s), discount γ
           ϵ, the maximum error allowed in the utility of any state
   local variables: U, U′, vectors of utilities for states in S, initially zero
                    δ, the maximum change in the utility of any state in an iteration

   repeat
       U ← U′; δ ← 0
       for each state s in S do
           U′[s] ← R(s) + γ max_{a ∈ A(s)} Σ_{s′} P(s′ | s, a) U[s′]
           if |U′[s] − U[s]| > δ then δ ← |U′[s] − U[s]|
   until δ < ϵ(1 − γ)/γ
   return U
```

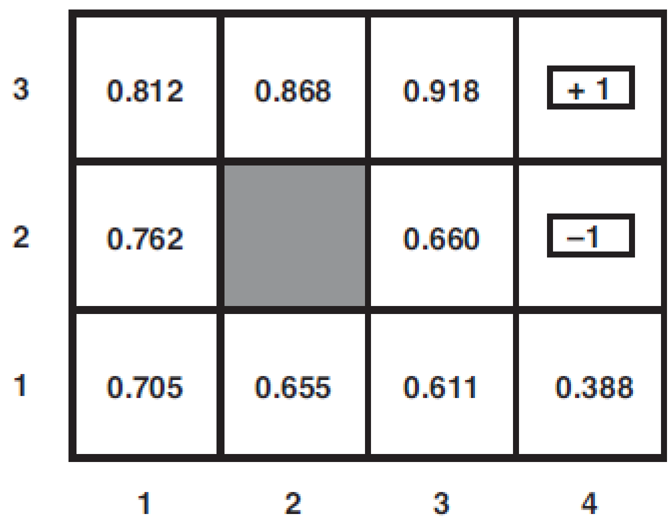
$$\boxed{U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']}$$

- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

The Value Iteration Algorithm

$$\boxed{U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')}$$

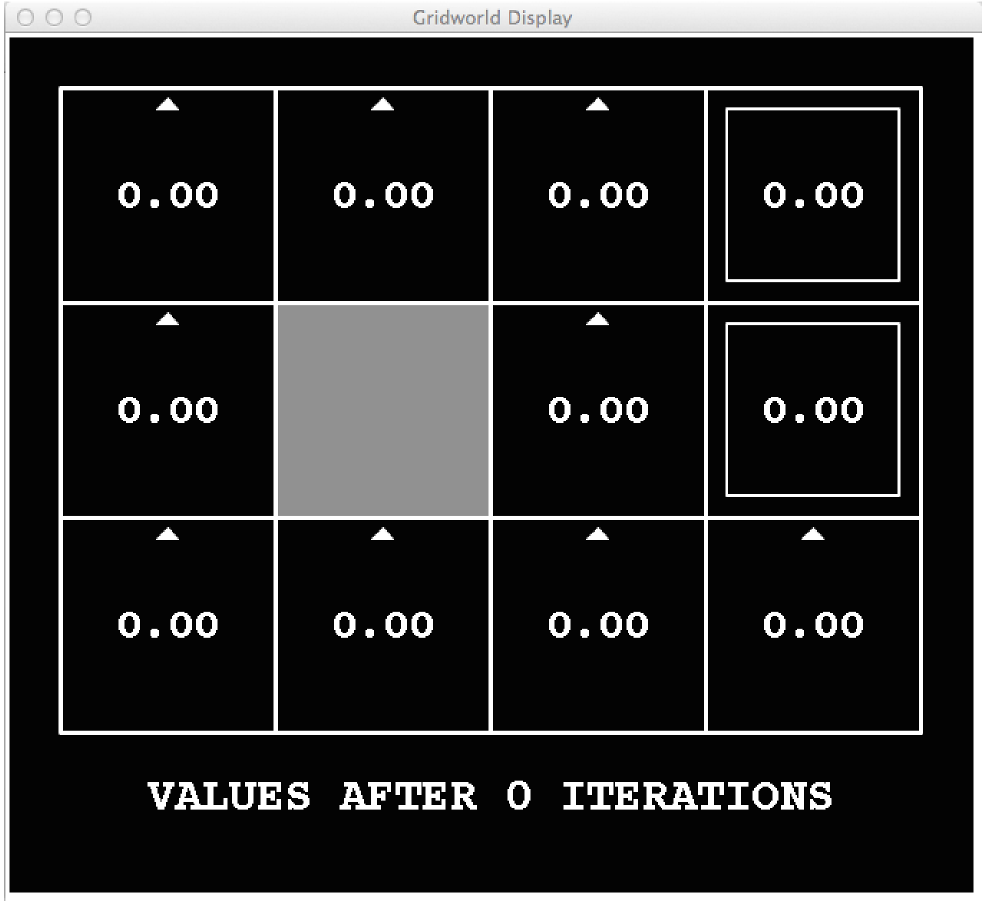
$$\begin{aligned} U(1,1) = -0.04 + \gamma \max[\; & 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), && (Up) \\ & 0.9U(1,1) + 0.1U(1,2), && (Left) \\ & 0.9U(1,1) + 0.1U(2,1), && (Down) \\ & 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)\;]. && (Right) \end{aligned}$$



$\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

Value Iteration

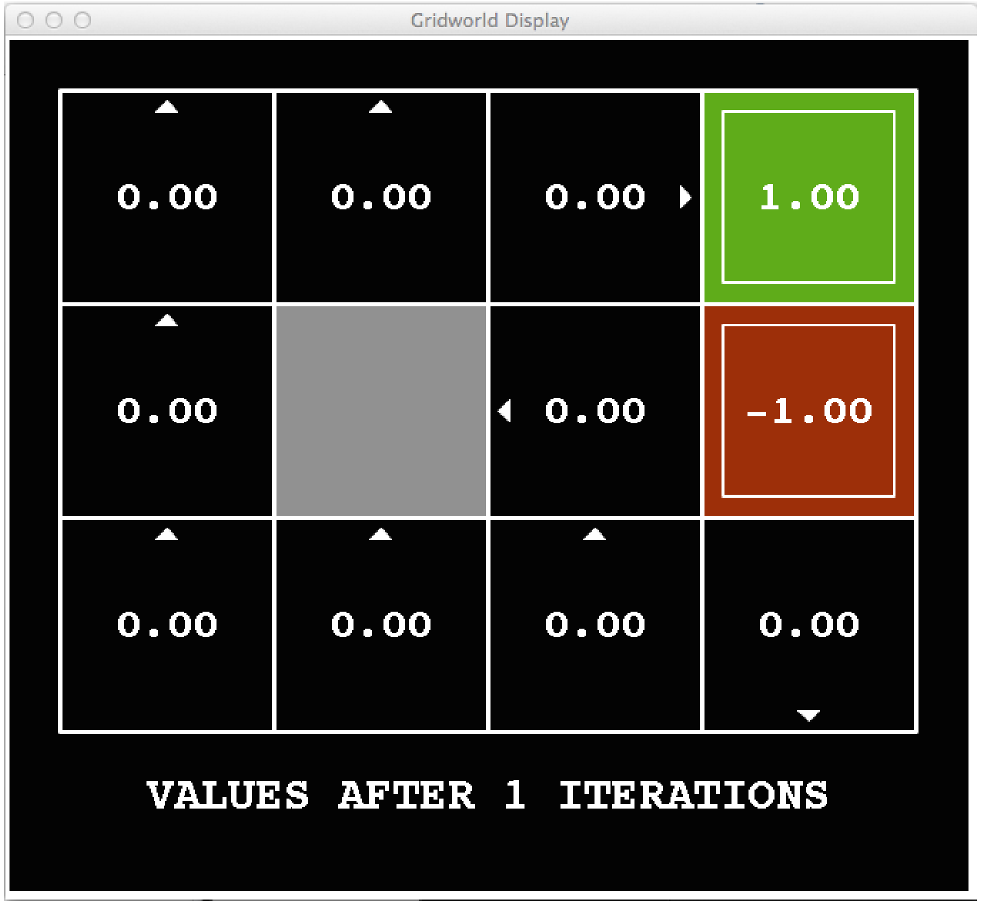
k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

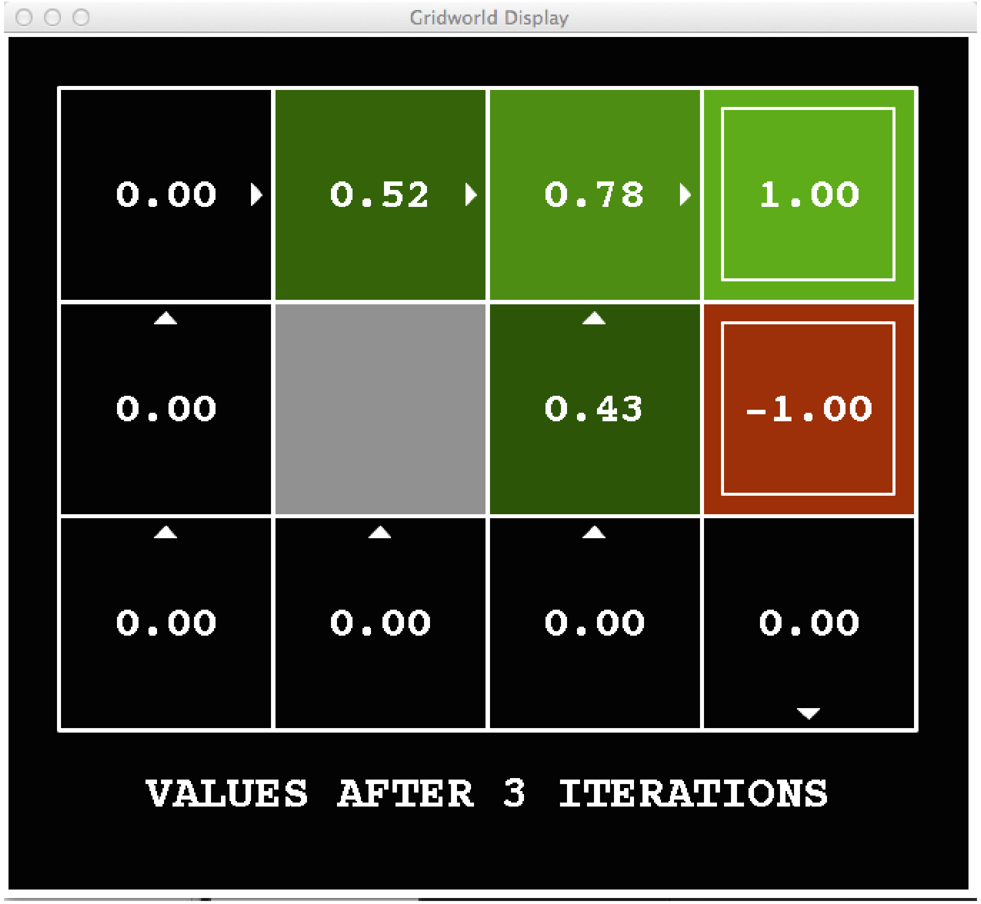
Value Iteration

k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

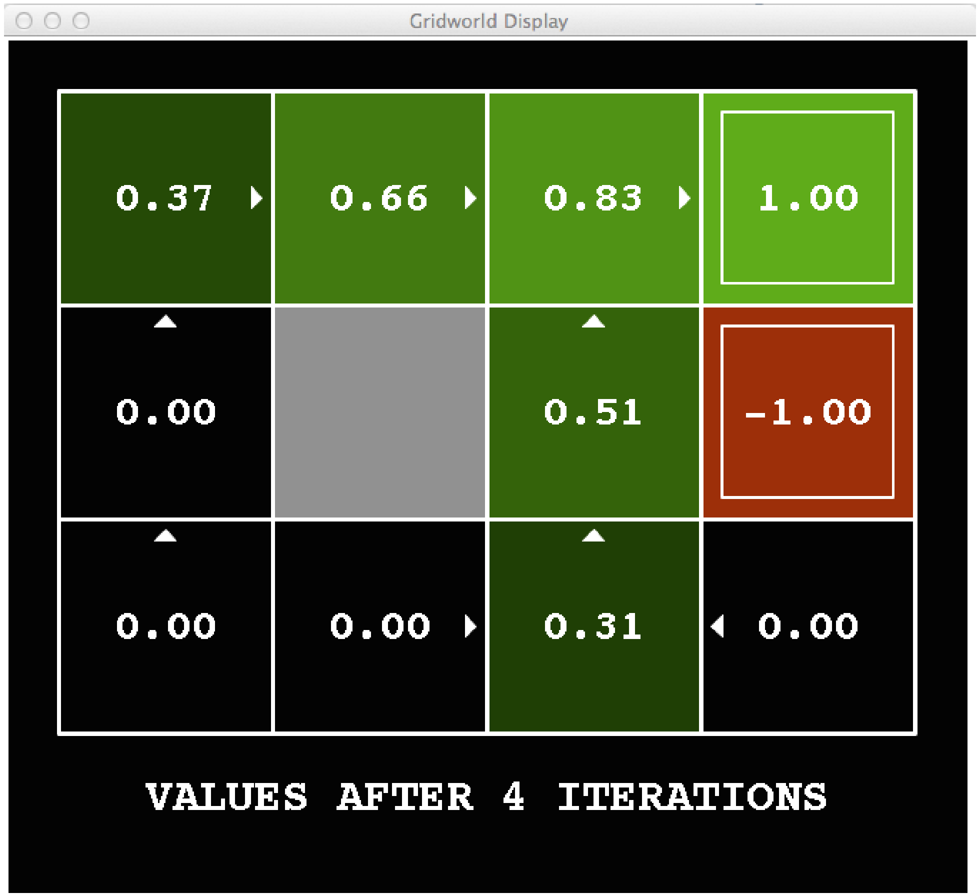
k=3



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=4



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=5

Noise = 0.2
Discount = 0.9
Living reward = 0

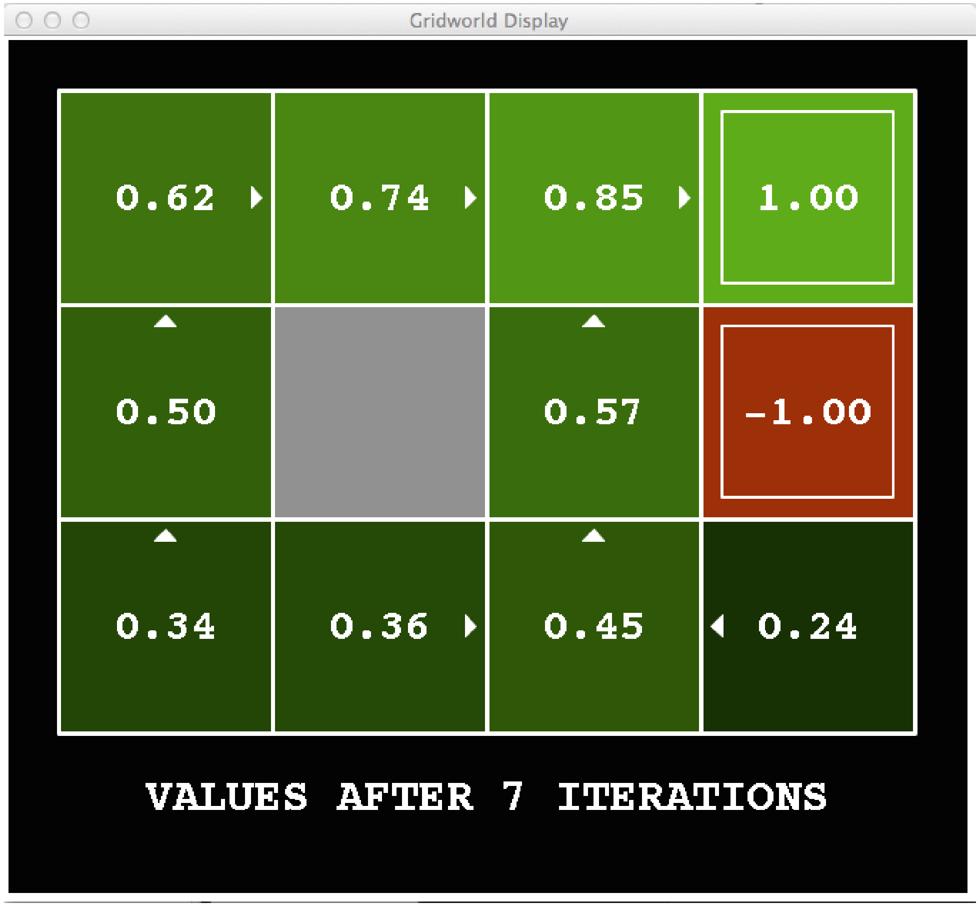
Value Iteration

k=6

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

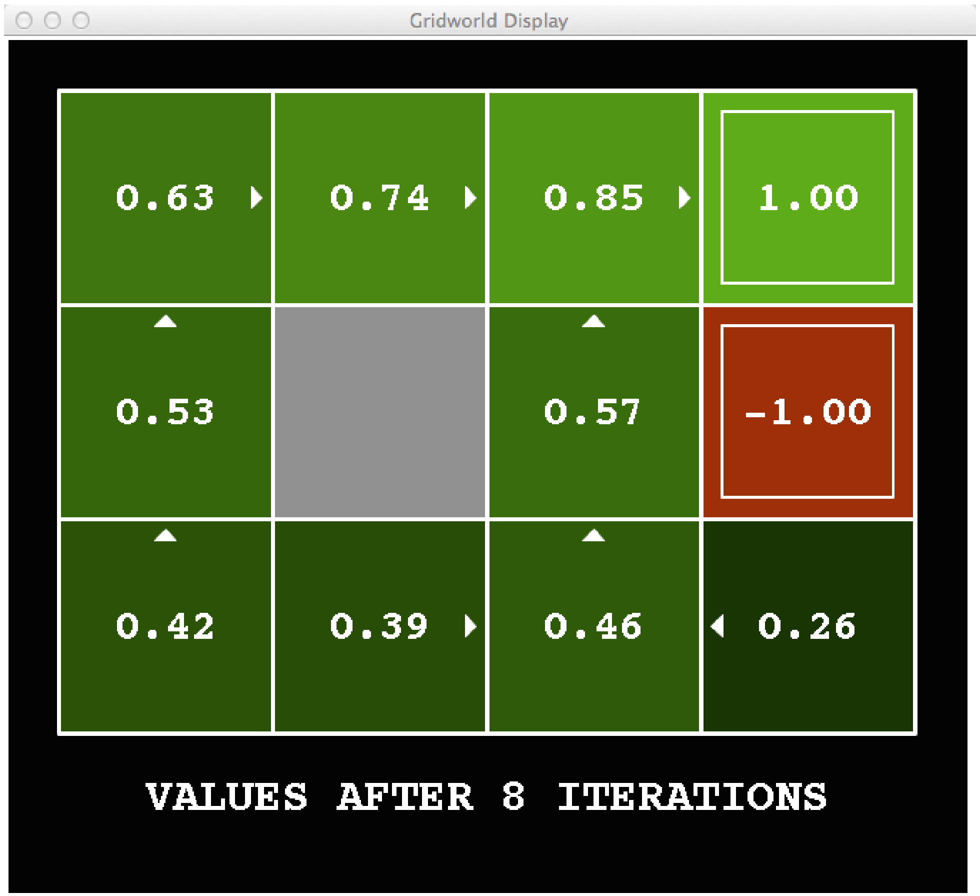
k=7



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

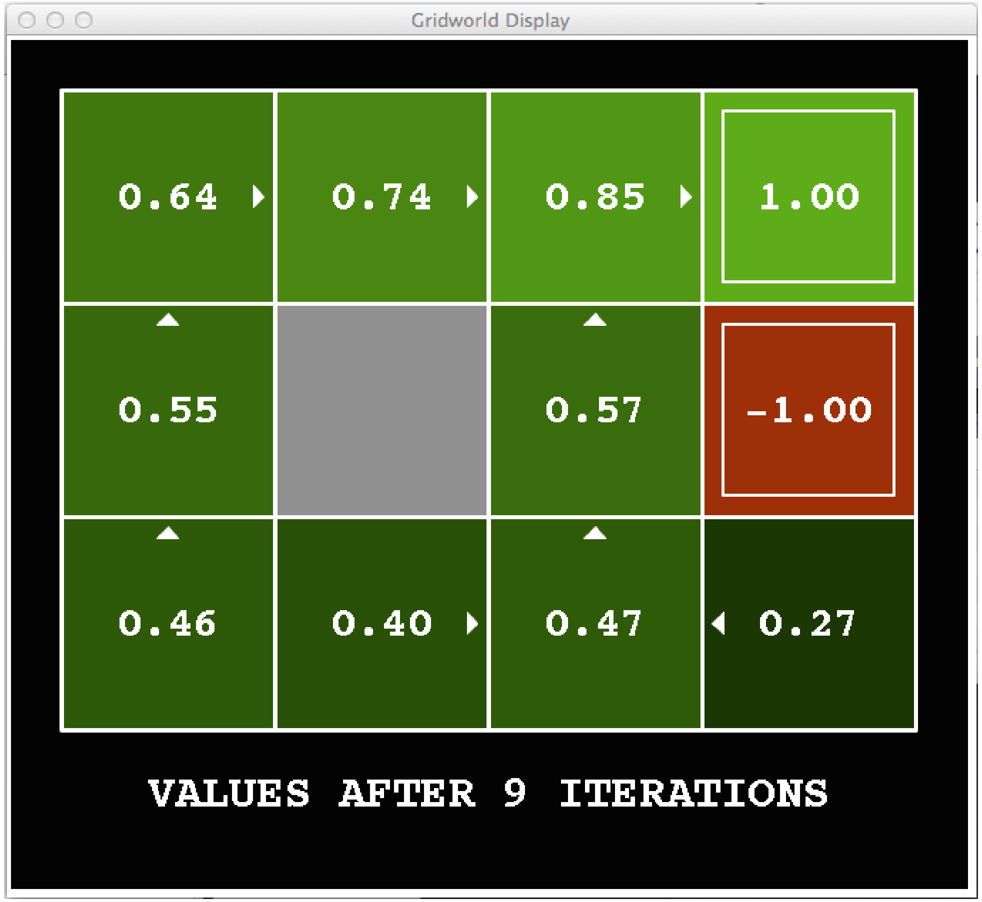
k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

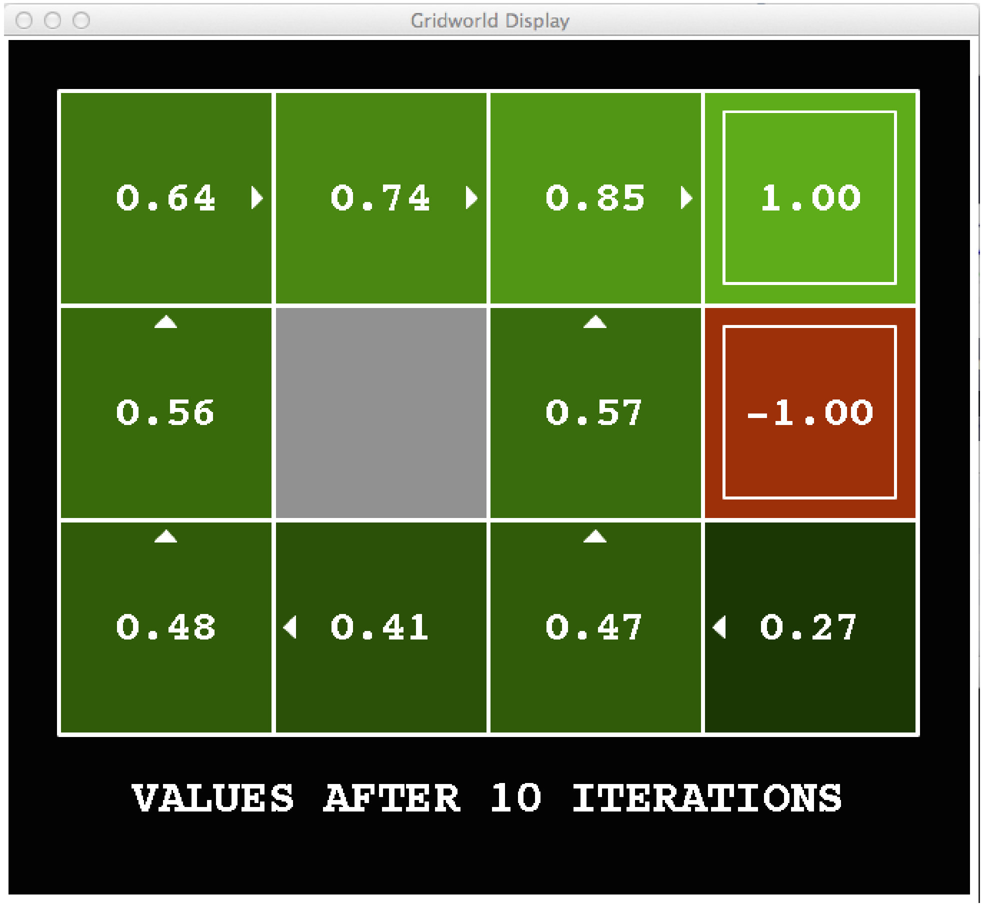
k=9



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

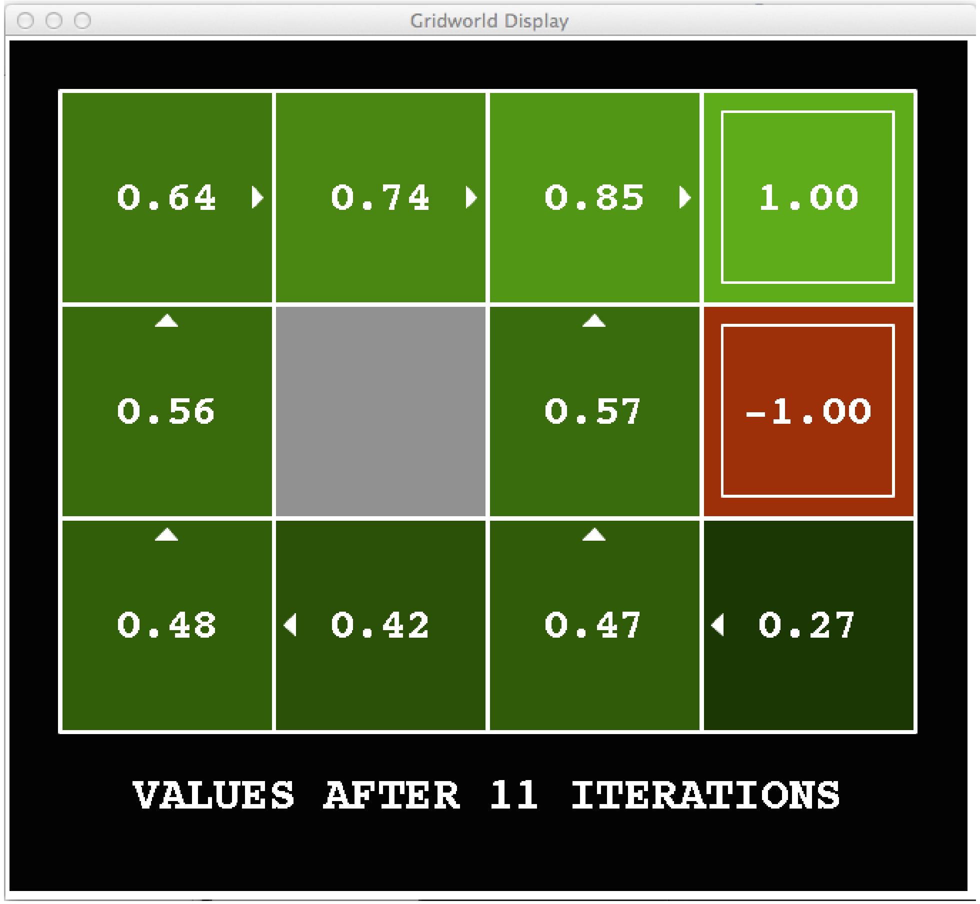
k=10



Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

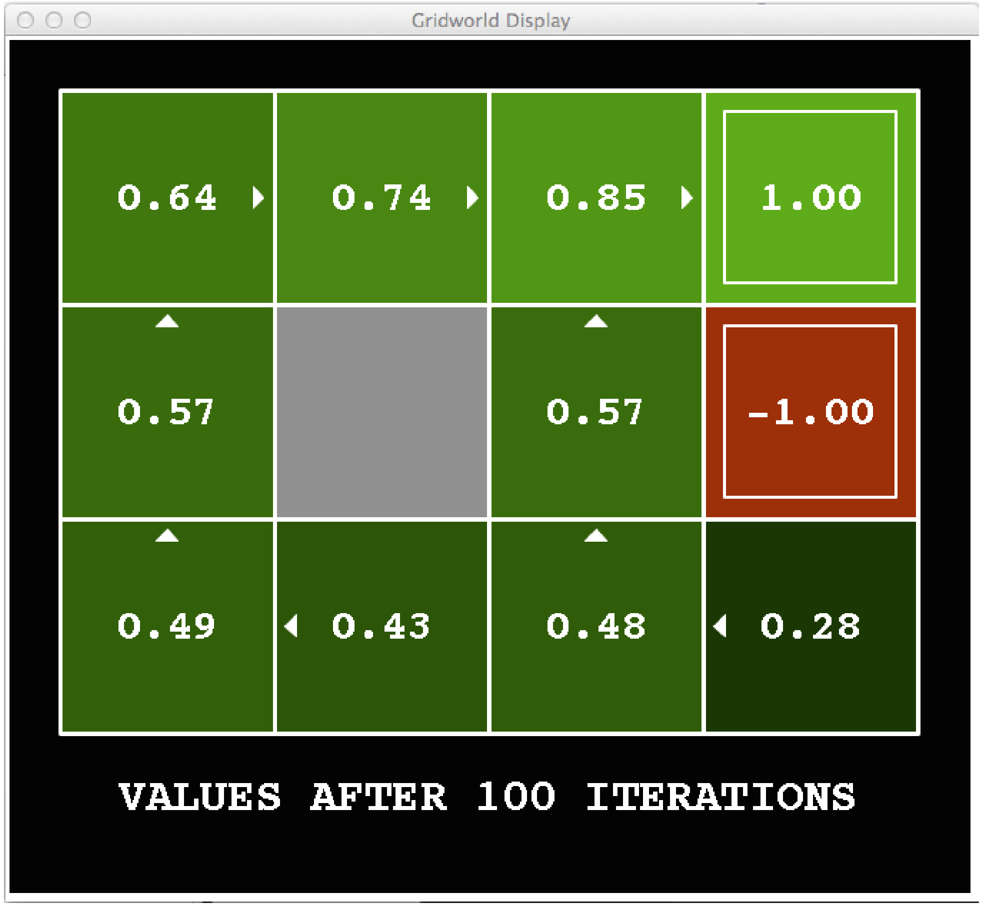
Value Iteration

k=12

Noise = 0.2
Discount = 0.9
Living reward = 0

Value Iteration

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

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:
  - It's slow – $O(S^2A)$ per iteration
  - The "max" at each state rarely changes
  - The policy often converges long before the values

Policy Iteration

- The policy iteration algorithm alternates the following two steps, beginning from some initial policy π_0:
  - Policy evaluation:
    - calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Policy improvement:
    - update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values

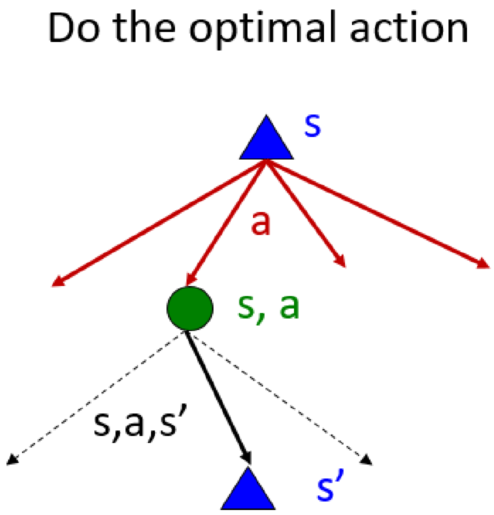
- Repeat steps until policy converges

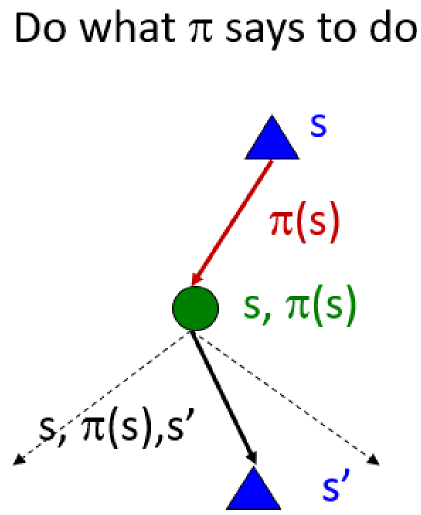
The Policy Iteration Algorithm

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\, U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

Policy Evaluation





- Expectimax trees 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
  - The tree's value would depend on which policy we fixed

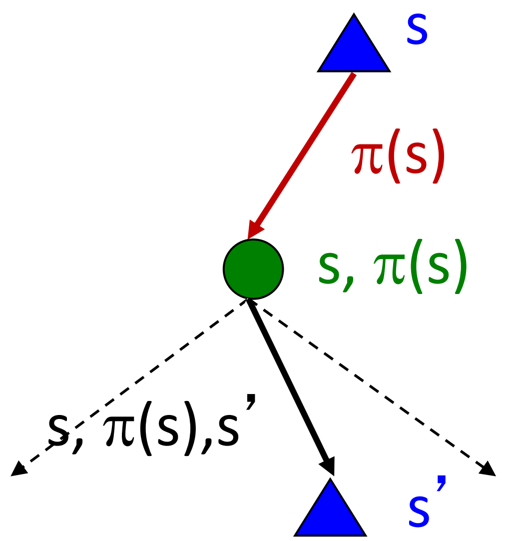
Policy Evaluation

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

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

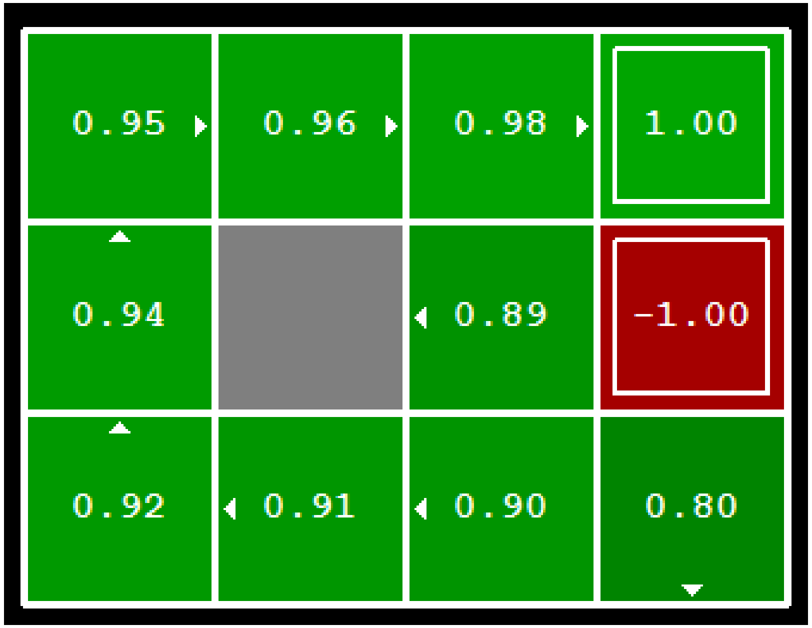
- Recursive relation (one-step look-ahead / Bellman equation):

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



Policy Extraction

- Computing actions from values
  - Let's imagine we have the optimal values V*(s)
  - How should we act?
    - It's not obvious!
  - We need to do one step



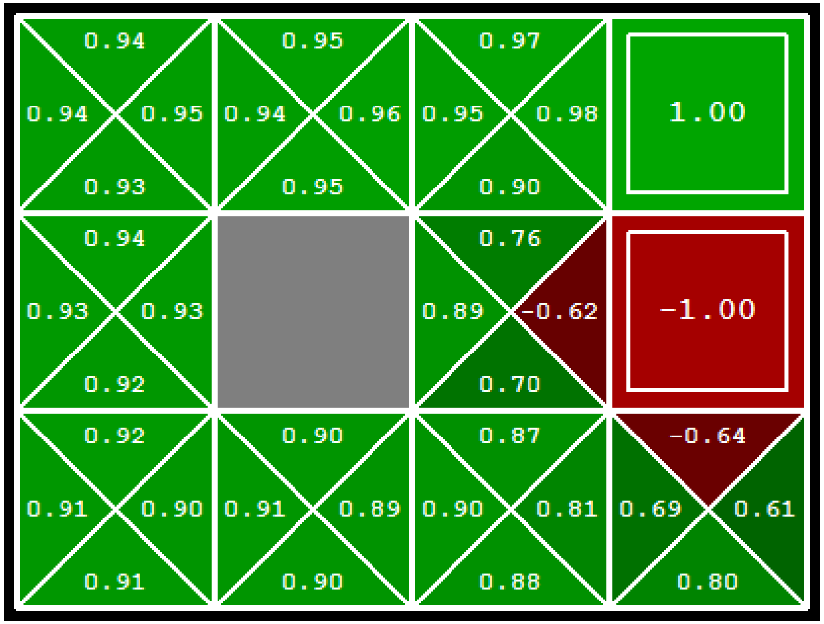
$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Policy Extraction

- Computing actions from values
  - Let's imagine we have the optimal q-values:
  - How should we act?
    - Completely trivial to decide!

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

  - Actions are easier to select from q-values than values!



Policy Iteration

- Evaluation: For fixed current policy π, find values with policy evaluation
  - Iterate until values converge：

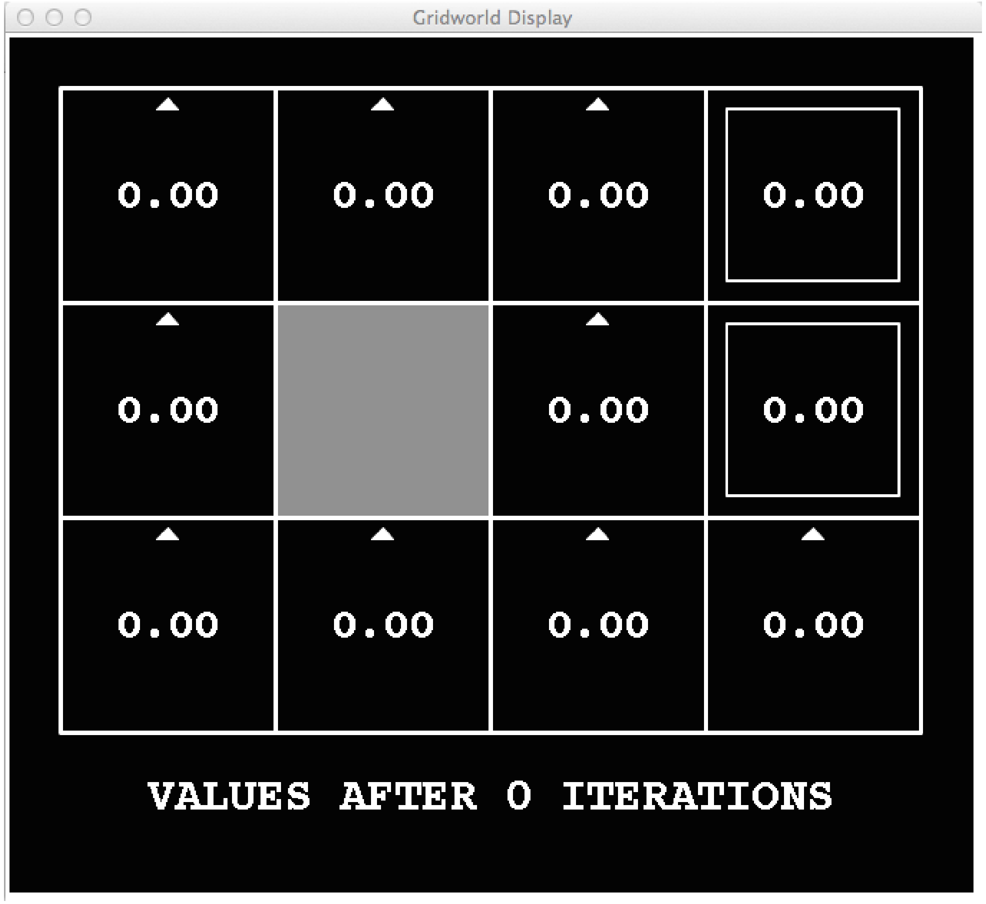
$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')\right]$$

- 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

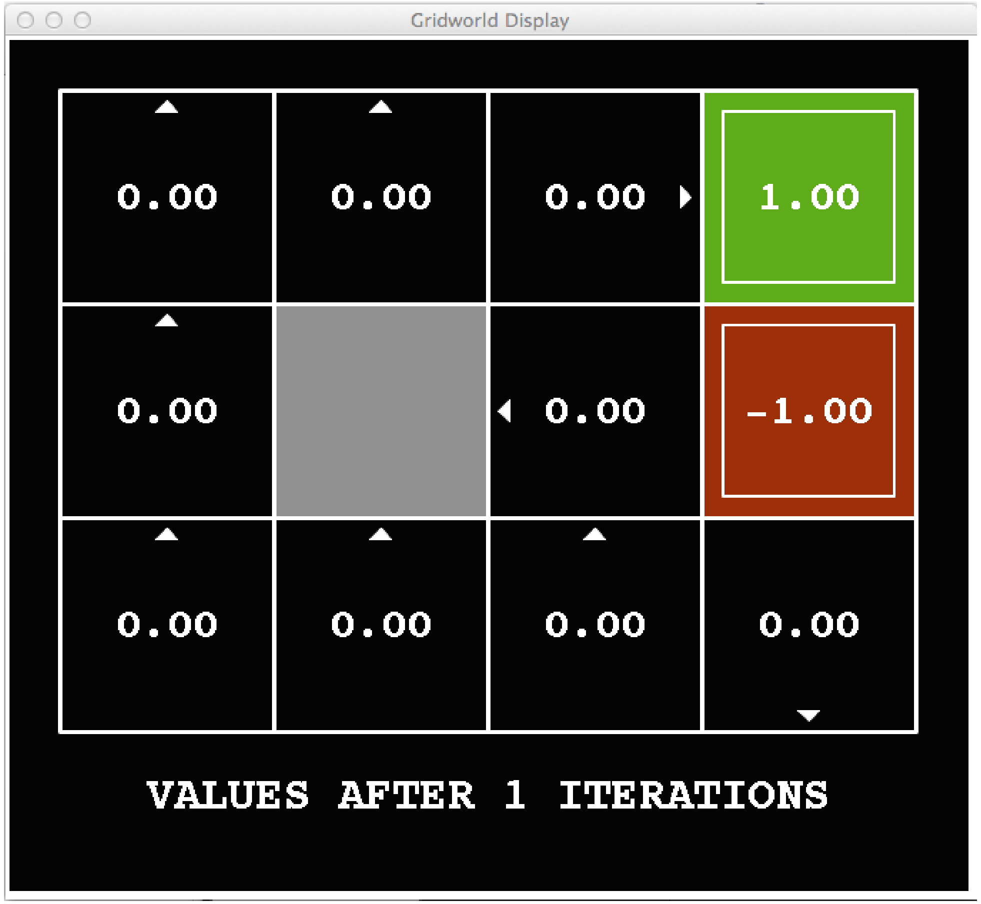
k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

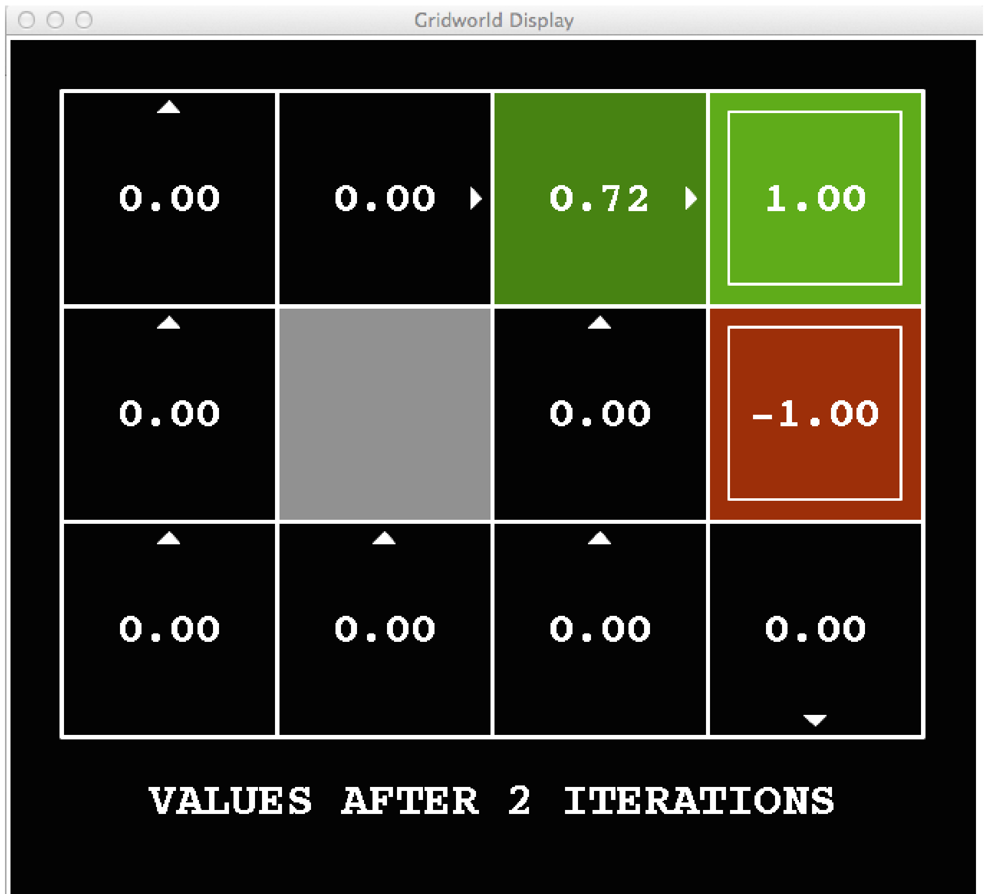
k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

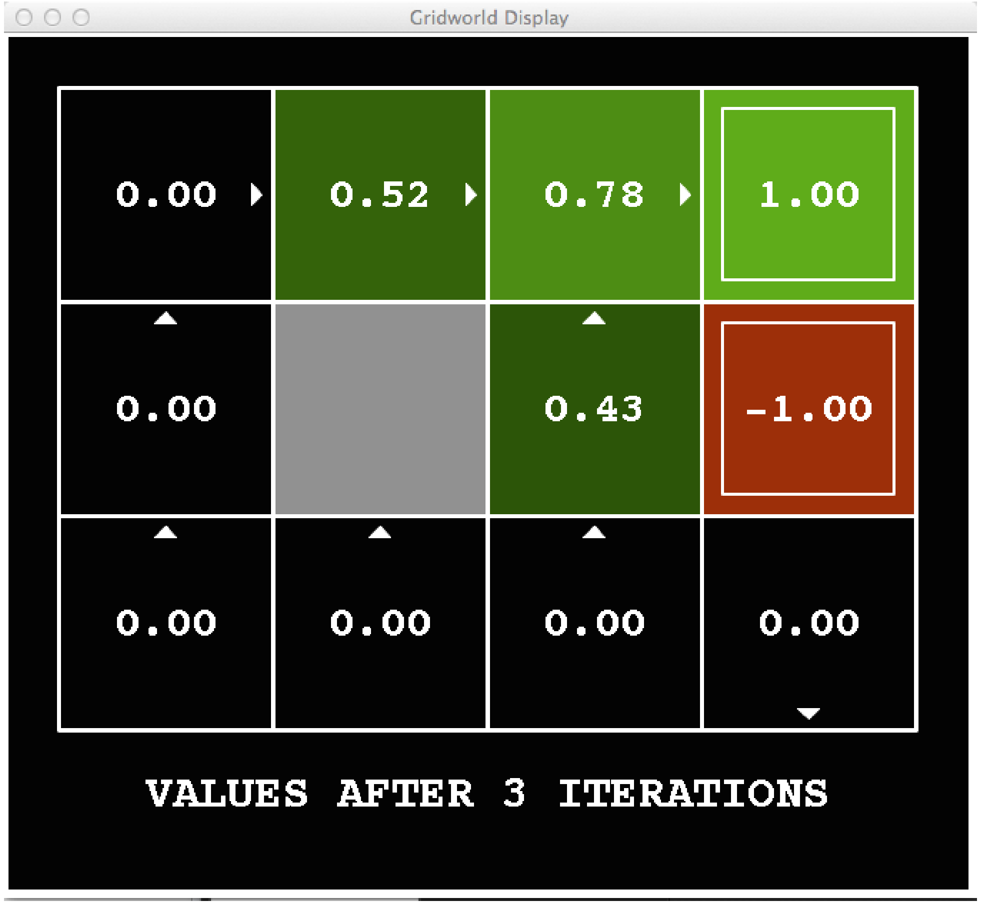
k=2



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

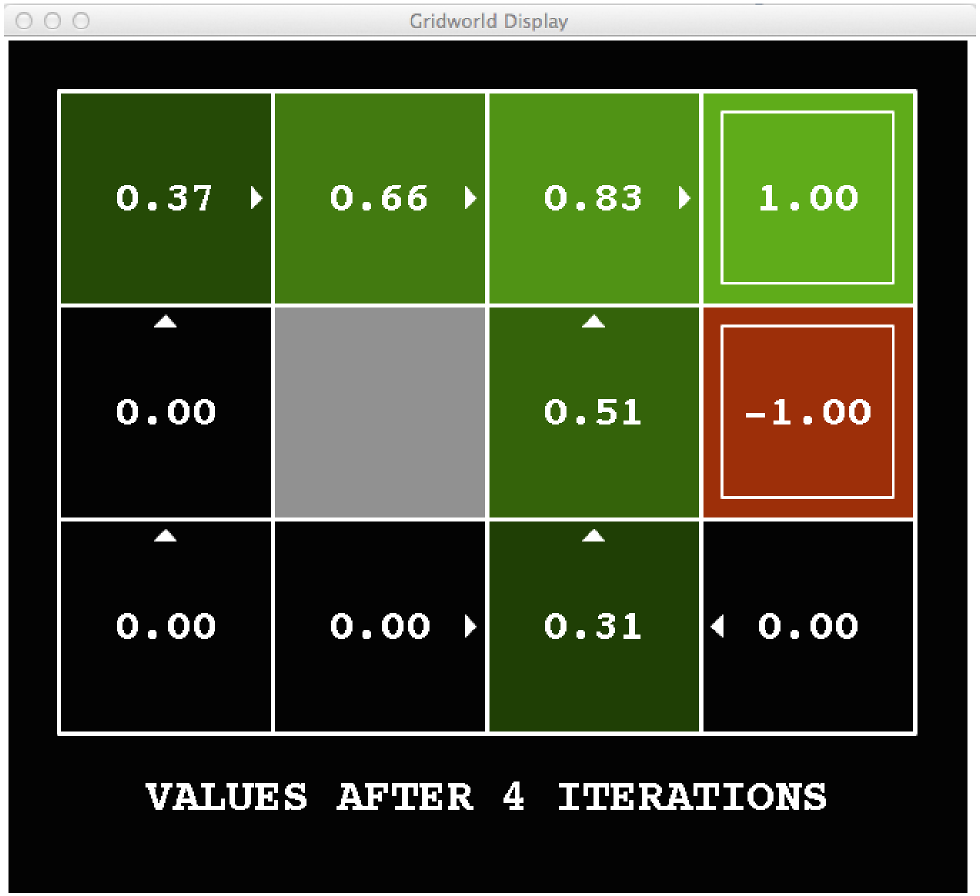
k=3



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

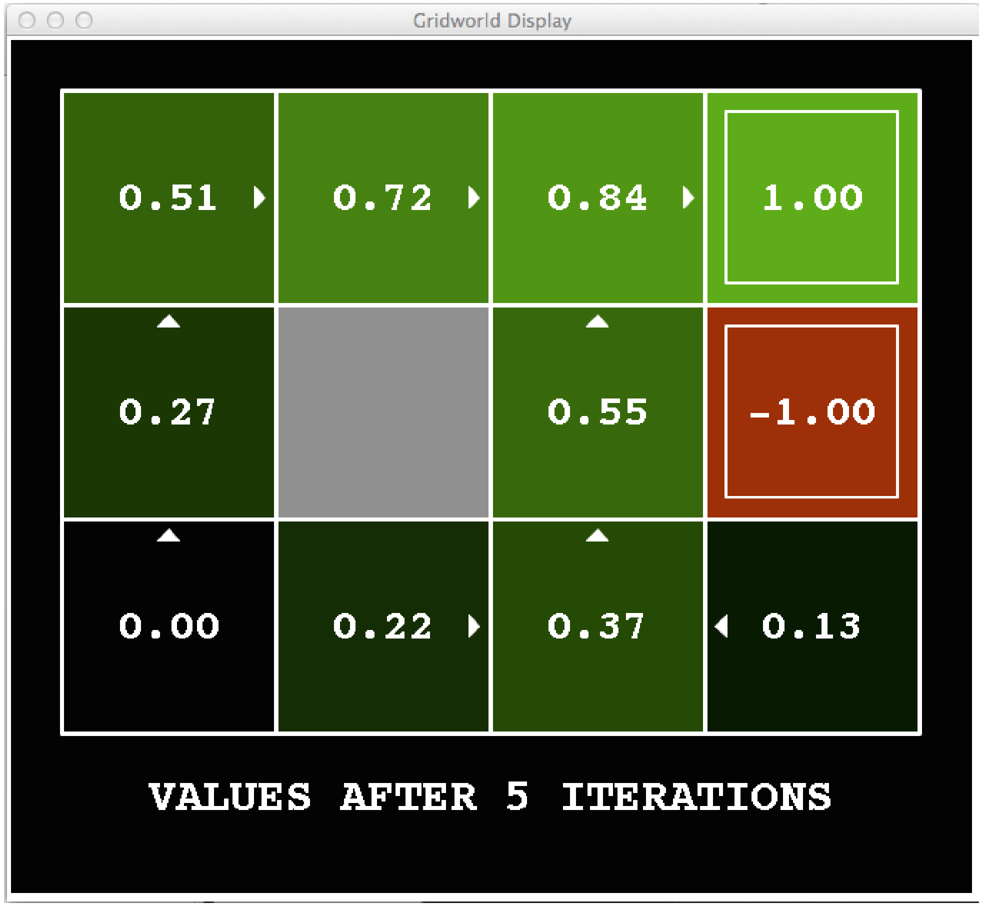
k=4



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

k=5



Noise = 0.2
Discount = 0.9
Living reward = 0

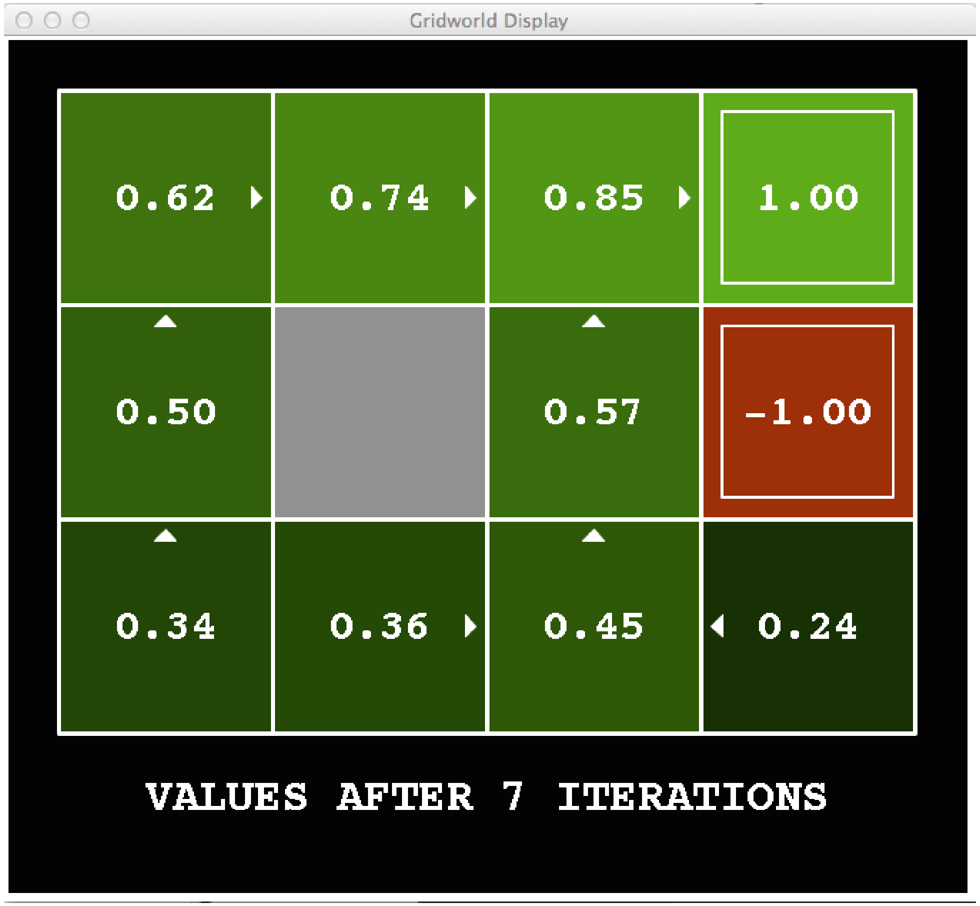
Policy Iteration

k=6

Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

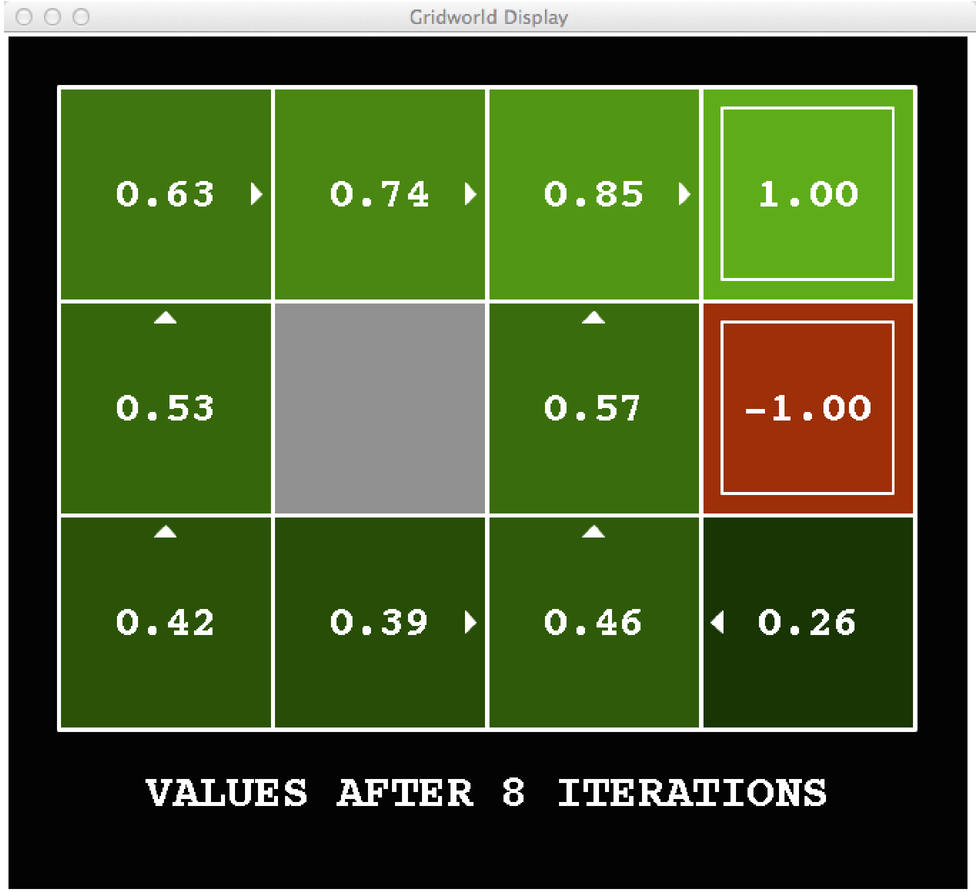
k=7



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

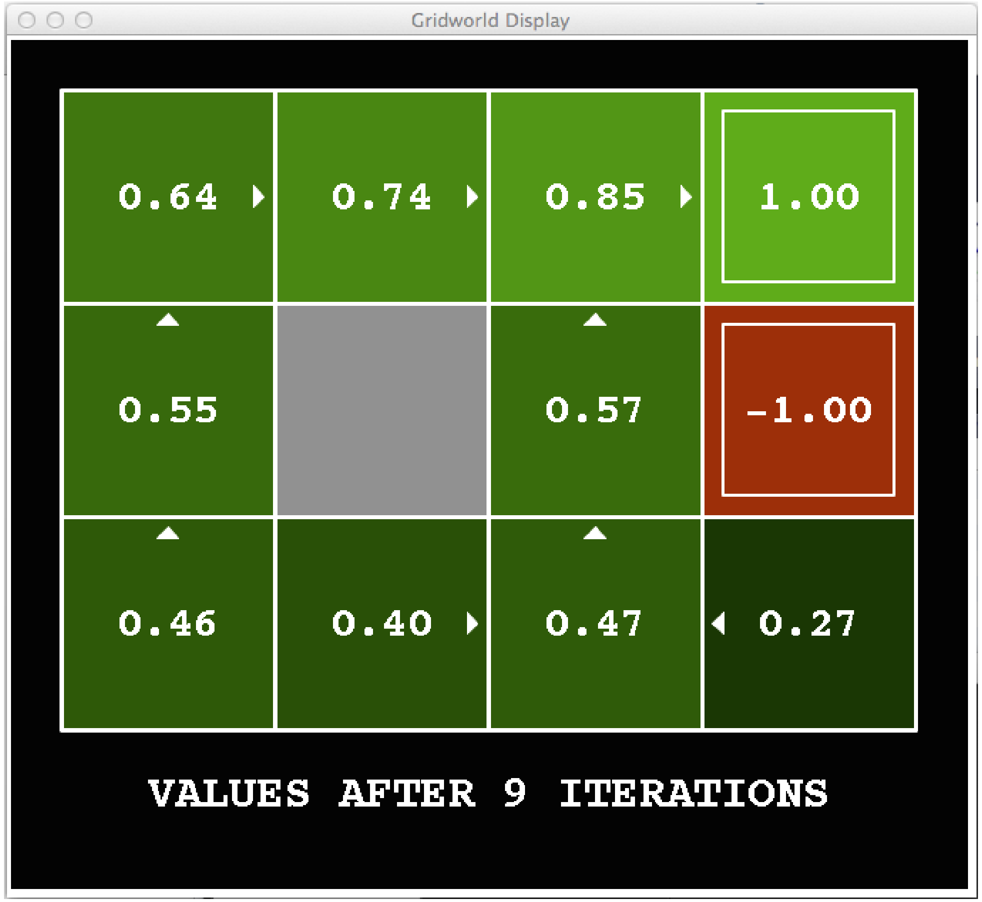
k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

k=9



Noise = 0.2
Discount = 0.9
Living reward = 0

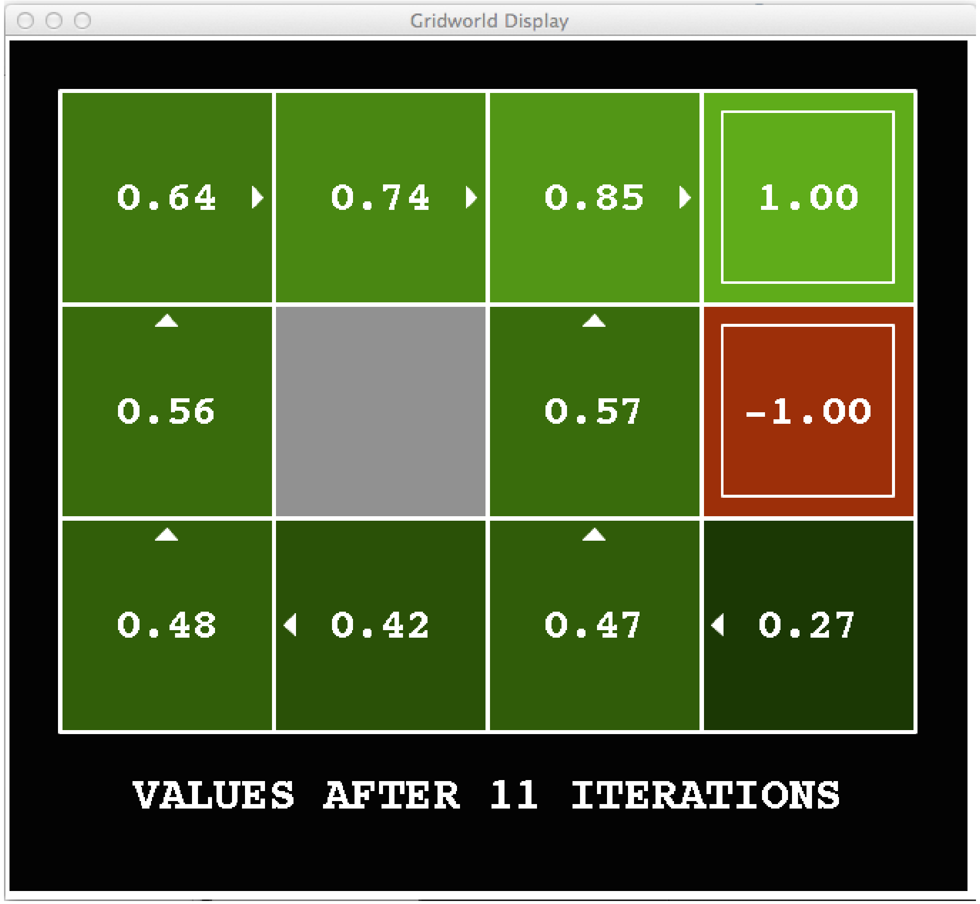
Policy Iteration

k=10

Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

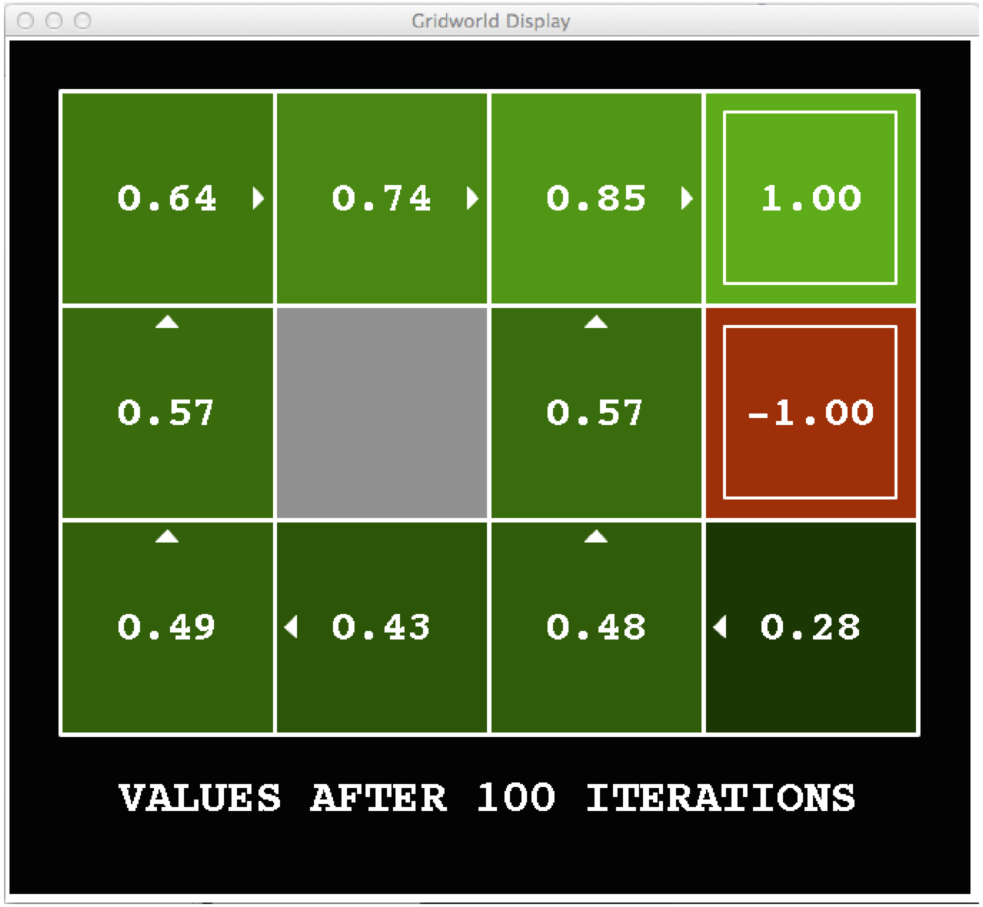
Policy Iteration

k=12

Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Comparison

- 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)

Summary

- 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 look ahead)

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

Readings

- Artificial Intelligence
  - Chapter 17.1-3

- Final Project
  - Due by June 6th, 2022

- Final Exam
  - June 21st, 2022