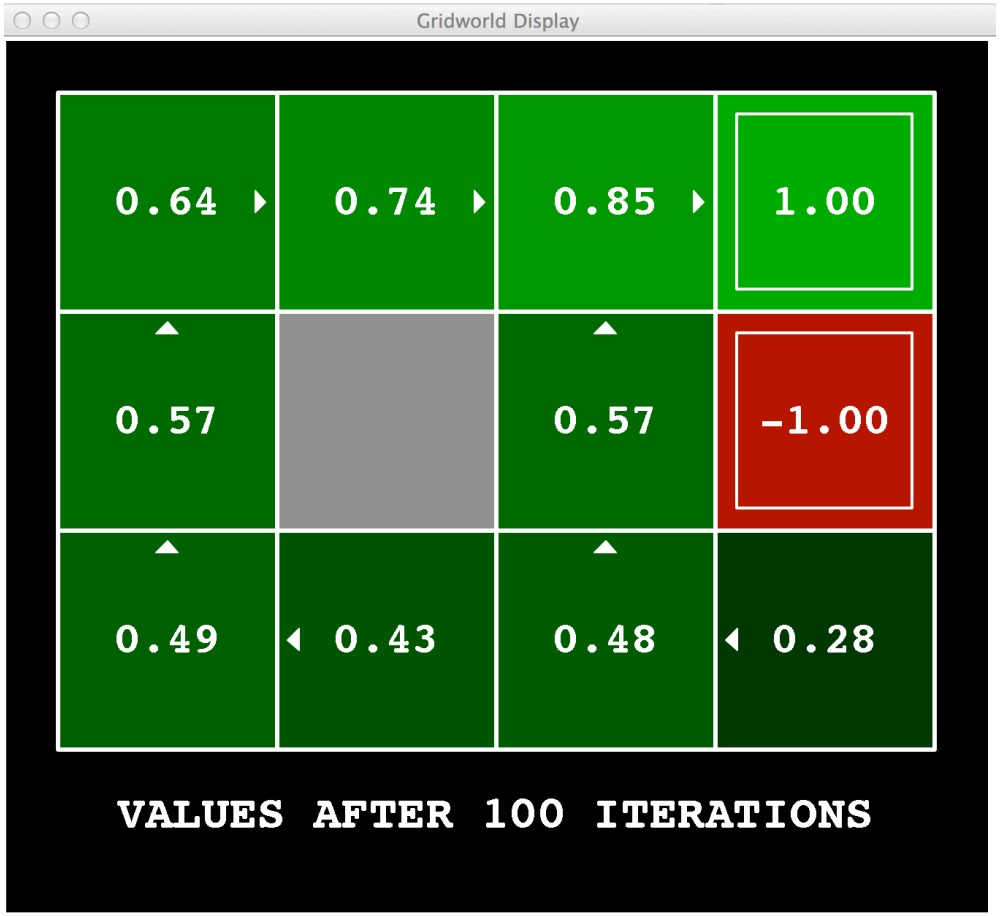
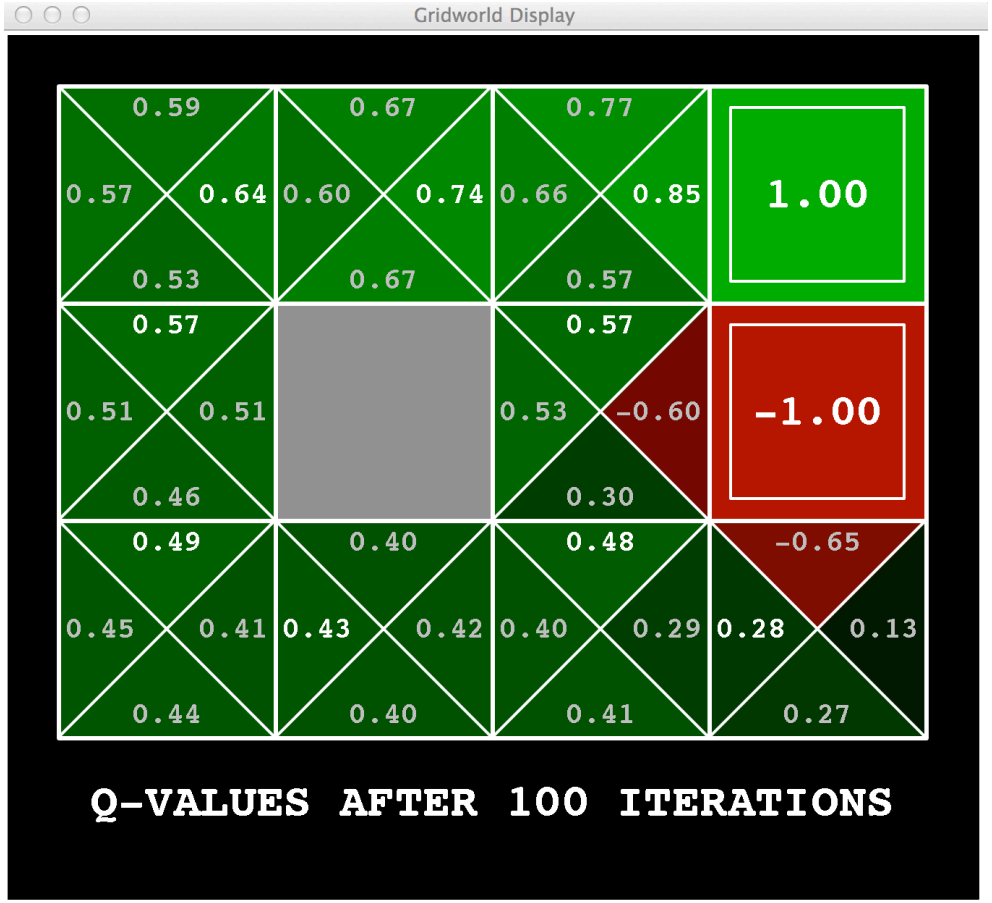
Markov Decision Processes: Solutions and Examples

With slides from Dan Klein and Pieter Abbeel and Percy Liang

Gridworld Values V*



Gridworld: Q*


Gridworld Display
0.59
0.57
0.64
0.53
0.67
0.60
0.74
0.67
0.77
0.66
0.85
0.57
1.00
0.57
0.51
0.51
0.46
0.57
0.53
-0.60
0.30
-1.00
0.49
0.45
0.41
0.44
0.40
0.43
0.42
0.40
0.48
0.40
0.29
0.41
-0.65
0.28
0.13
0.27
Q-VALUES AFTER 100 ITERATIONS

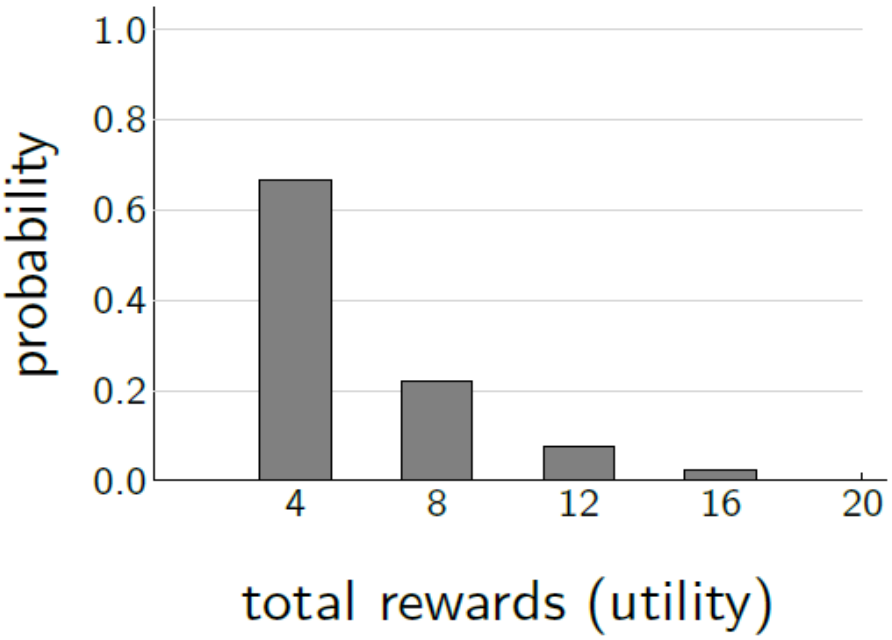
Example: Quit/Stay Game

For each round $r = 1, 2, \ldots$

- You choose stay or quit.
- If quit, you get $\$10$ and we end the game.
- If stay, you get $\$4$ and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.

Total Winnings (Reward): Policy 1

If follow policy "stay":



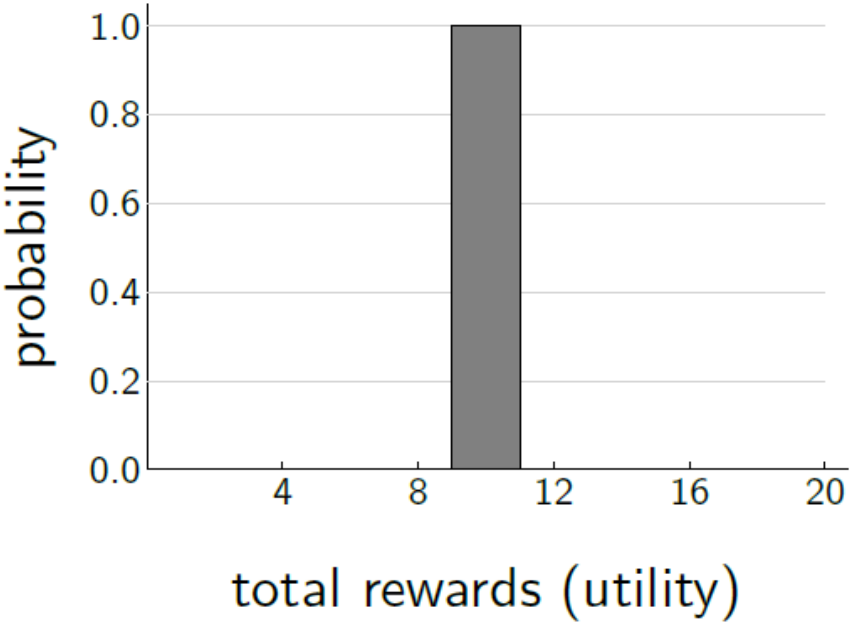
a1 / (1−r) =
4 / (1-2/3)

Expected utility:

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \cdots = 12$$

Total Winnings (Reward): Policy 2

If follow policy "quit":



Expected utility:

$$1(10) = 10$$

Reminder: Probabilities Sum to 1

Example: transition probabilities

s	a	s'	$T(s,a,s')$
in	quit	end	1
in	stay	in	$2/3$
in	stay	end	$1/3$

For each state s and action a:

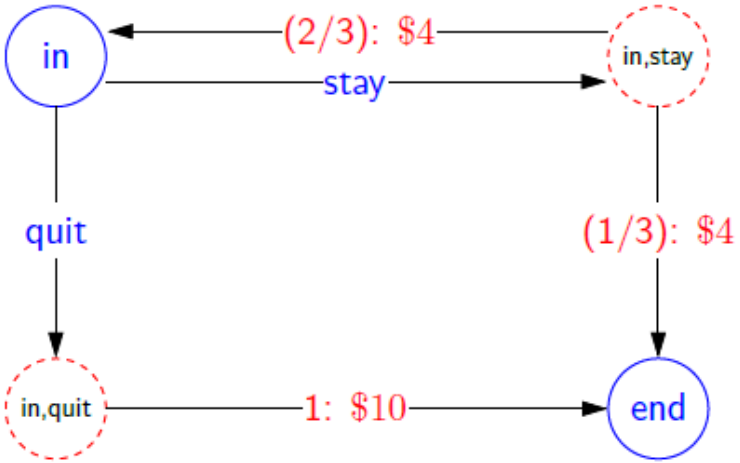
$$\sum_{s' \in \text{States}} T(s,a,s') = 1$$

Successors: s' such that $T(s,a,s') > 0$

MDP **DIAGRAM** for Game 1

For each round $r = 1, 2, \ldots$

- You choose stay or quit.
- If quit, you get $\$10$ and we end the game.
- If stay, you get $\$4$ and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.



Recycling Robot


news.bbc.co.uk

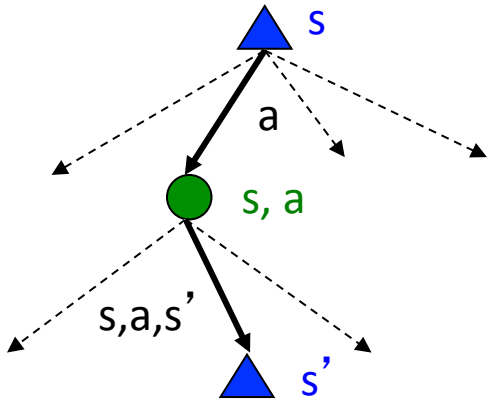
- MDP setup:
  - **Actions**: **search** for trashcan, wait for someone to bring a trashcan, or go home and recharge battery
  - **States**: two energy levels – high and low
  - Searching runs down battery, waiting does not, and a depleted battery has a very low reward

MDP Robot: Transition Probabilities

$s = s_t$	$s' = s_{t+1}$	$a = a_t$	$P^a_{ss'}$	$R^a_{ss'}$
high	high	search	α	R^{search}
high	low	search	$1 - \alpha$	R^{search}
low	high	search	$1 - \beta$	-3
low	low	search	β	R^{search}
high	high	wait	1	R^{wait}
high	low	wait	0	R^{wait}
low	high	wait	0	R^{wait}
low	low	wait	1	R^{wait}
low	high	recharge	1	0
low	low	recharge	0	0

Recap: 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



- 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)

Recap: Optimal Quantities

- The value (utility) of a state s:

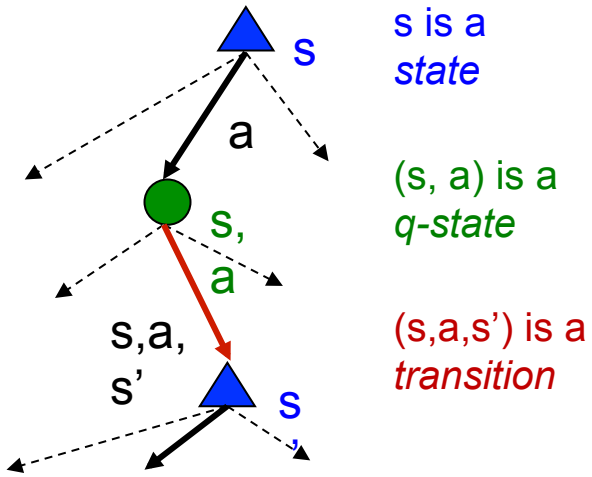
  $V^*(s)$ = expected utility starting in s and acting optimally

- 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



[Demo: gridworld values (L9D1)]

Solution to an MDP = Policy π

- Gives the action to take from a given state regardless of history
- Two arrays indexed by state
  - V is the value function, namely the discounted sum of rewards on average from following a policy
  - π is an array of actions to be taken in each state (**Policy**)

2 basic steps

$$\pi(s) := \arg\max_a \sum_{s'} P_a(s,s')V(s')$$

$$V(s) := R(s) + \gamma \sum P_{\pi(s)}(s,s')V(s')$$

Recap: **Discounting**

Discount $\gamma < 1$:

[stay, stay, stay, stay]: $4 + \gamma \cdot 4 + \gamma^2 \cdot 4 + \gamma^3 \cdot 4$

Discount $\gamma = 1$ (save for the future):

[stay, stay, stay, stay]: $4 + 4 + 4 + 4 = 16$

Discount $\gamma = 0$ (live in the moment):

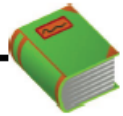
[stay, stay, stay, stay]: $4 + 0 \cdot (4 + \cdots) = 4$

Interpretation: small γ favors rewards sooner than later

Recap: Solving MDPs

Search problem: path (sequence of actions)

MDP:



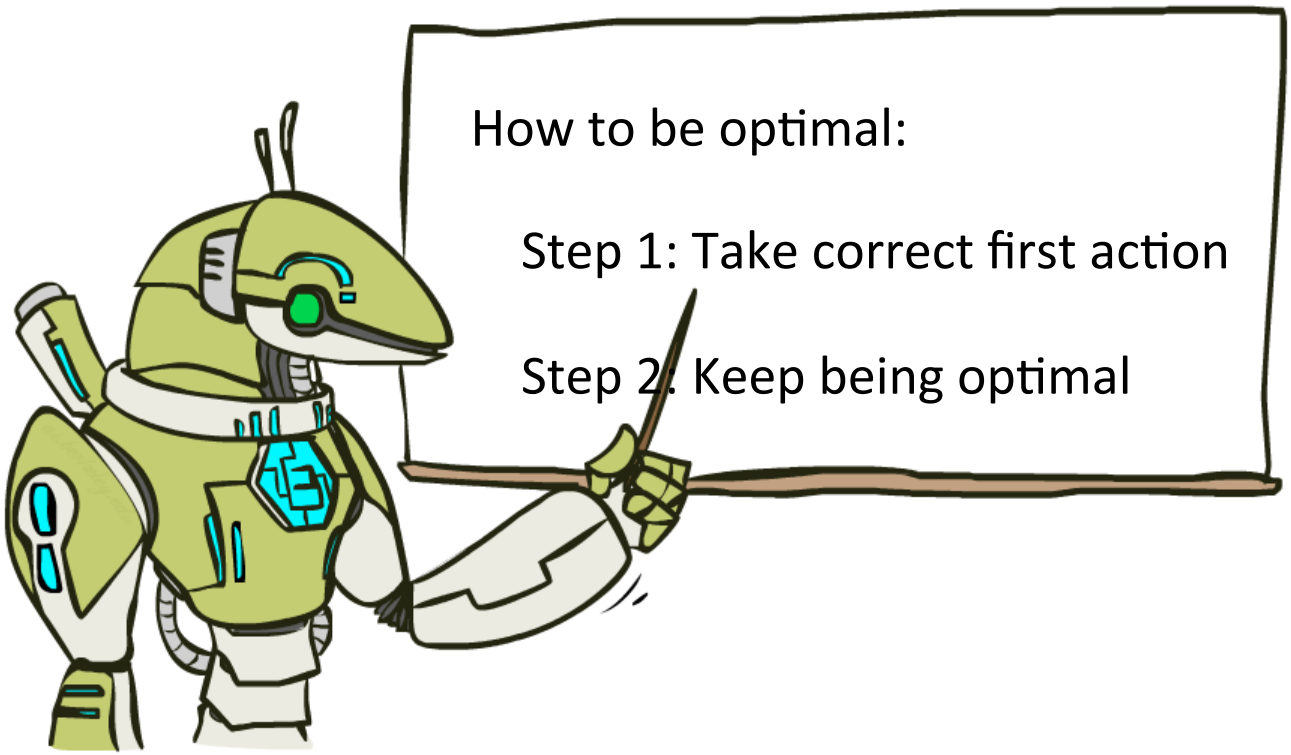
Definition: policy

A **policy** π is a mapping from each state $s \in$ States to an action $a \in$ Actions(s).

Example: game 2

s	$\pi(s)$
in	stay
odd	quit
end	-

The Bellman Equations



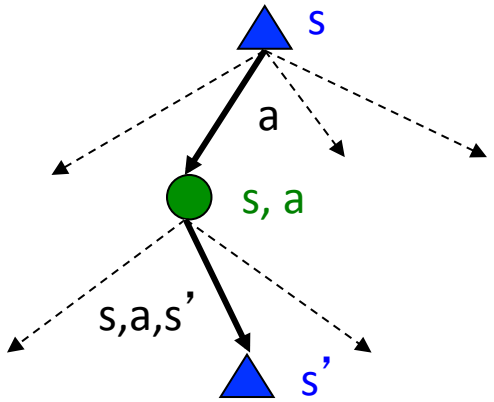
The Bellman Equations

- Definition of "optimal utility" via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

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

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

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



- These are the Bellman equations, and they characterize optimal values in a way we'll use over and over

Value Iteration

Goal: try to get directly at maximum expected utility

Definition: optimal value

The **optimal value** V_{opt} is the maximum value attained by any policy.

Value Iteration

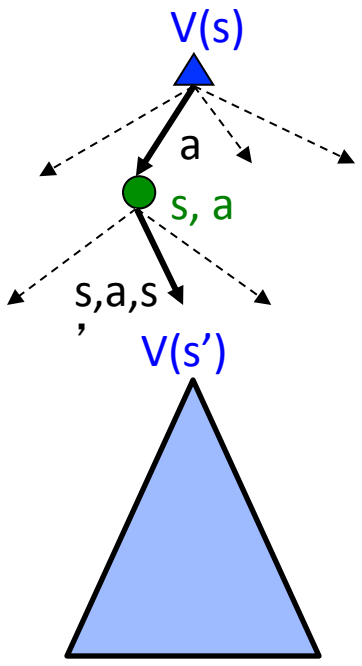
- Bellman equations characterize the optimal values:

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

- Value iteration computes them:

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

- Value iteration is just a fixed point solution method
  - … though the V_k vectors are also interpretable as time-limited values



Value Iteration: Algorithm

Algorithm: value iteration [Bellman, 1957]

Initialize $V_{\text{opt}}^{(0)}(s) \leftarrow 0$ for all states s.

For iteration $t = 1, \ldots, t_{\text{VI}}$:

For each state s:

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s,a)}$$

Value Iteration Example: Game 1

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	0.00	(t=0 iterations)
$\pi_{\text{opt}}(s)$	-	-	

For each state s:

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s,a)}$$

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	10.00	(t=1)
$\pi_{\text{opt}}(s)$	-	**quit**	

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	10.67	(t=2)
$\pi_{\text{opt}}(s)$	-	**stay**	

Value Iteration Example: Game 1 (cont'd)

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	10.67	(t=2)
$\pi_{\text{opt}}(s)$	-	stay	

For each state s:

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s,a)}$$

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	11.11	(t=3)
$\pi_{\text{opt}}(s)$	-	stay	

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	12.00	(t=100)
$\pi_{\text{opt}}(s)$	-	stay	

Value Iteration Example: Game 1

Policy evaluation with π is "stay":

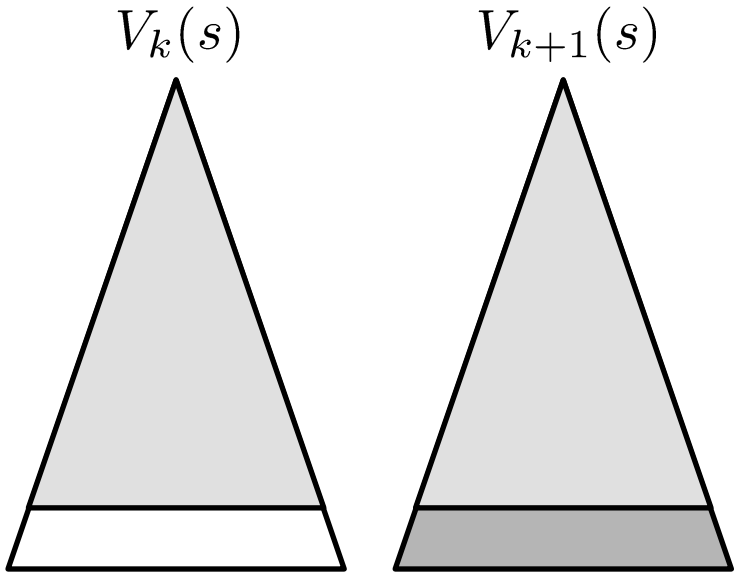
s	end	in	
$V_\pi^{(t)}$	0.00	12.00	($t = 100$ iterations)

Value iteration:

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	12.00	($t = 100$ iterations)
$\pi_{\text{opt}}(s)$	-	stay	

Convergence*

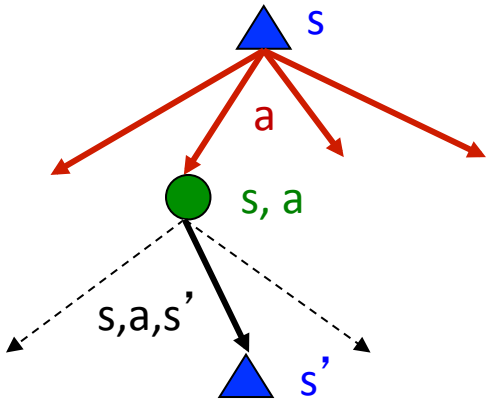
- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
  - That last layer is at best all R_{MAX}
  - It is at worst R_{MIN}
  - But everything is discounted by γ^k that far out
  - So V_k and V_{k+1} are at most γ^k max|R| different
  - So as k increases, the values converge



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

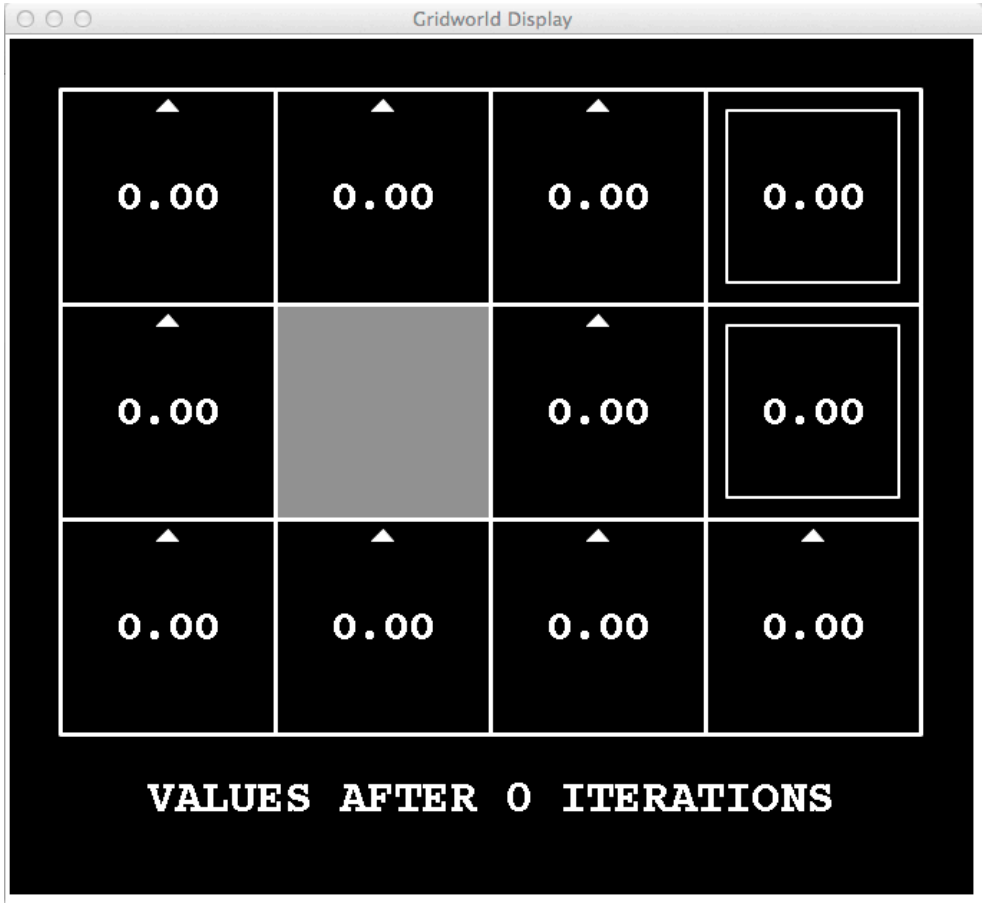
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$



- Problem 1: It's slow – $O(S^2A)$ per iteration
- Problem 2: The "max" at each state rarely changes
- Problem 3: The policy often converges long before the values

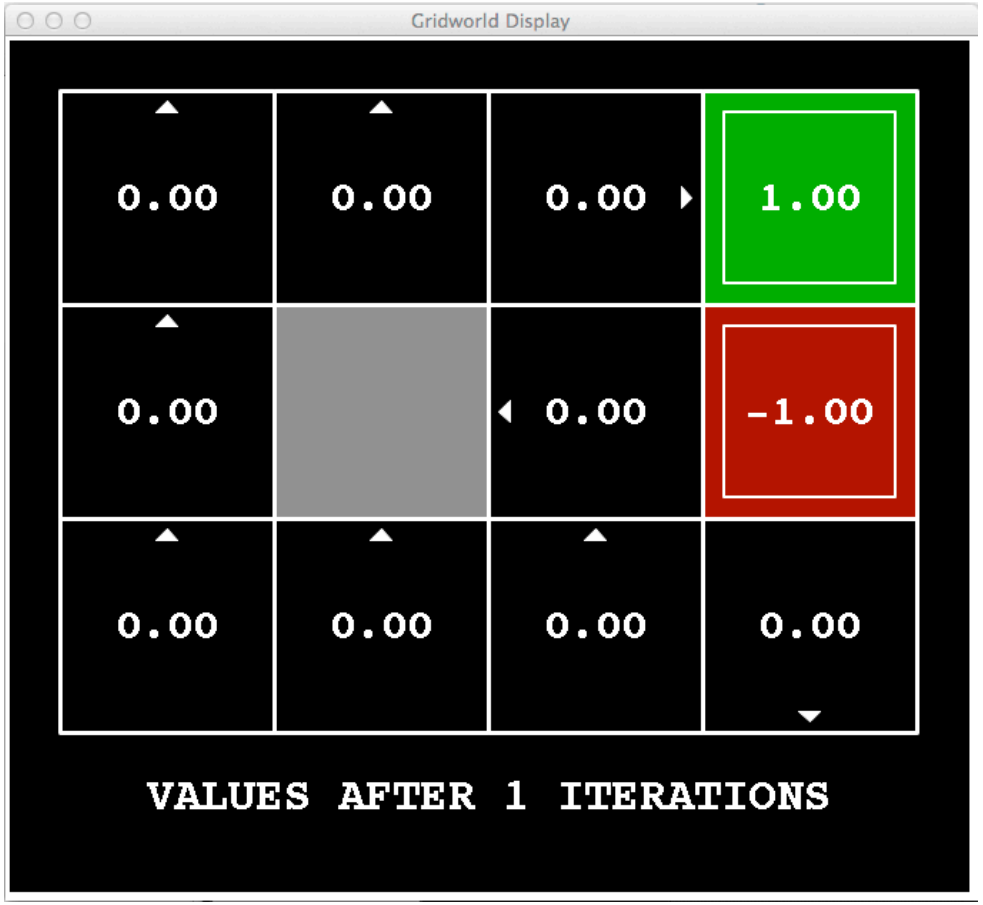
[Demo: value iteration (L9D2)]

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

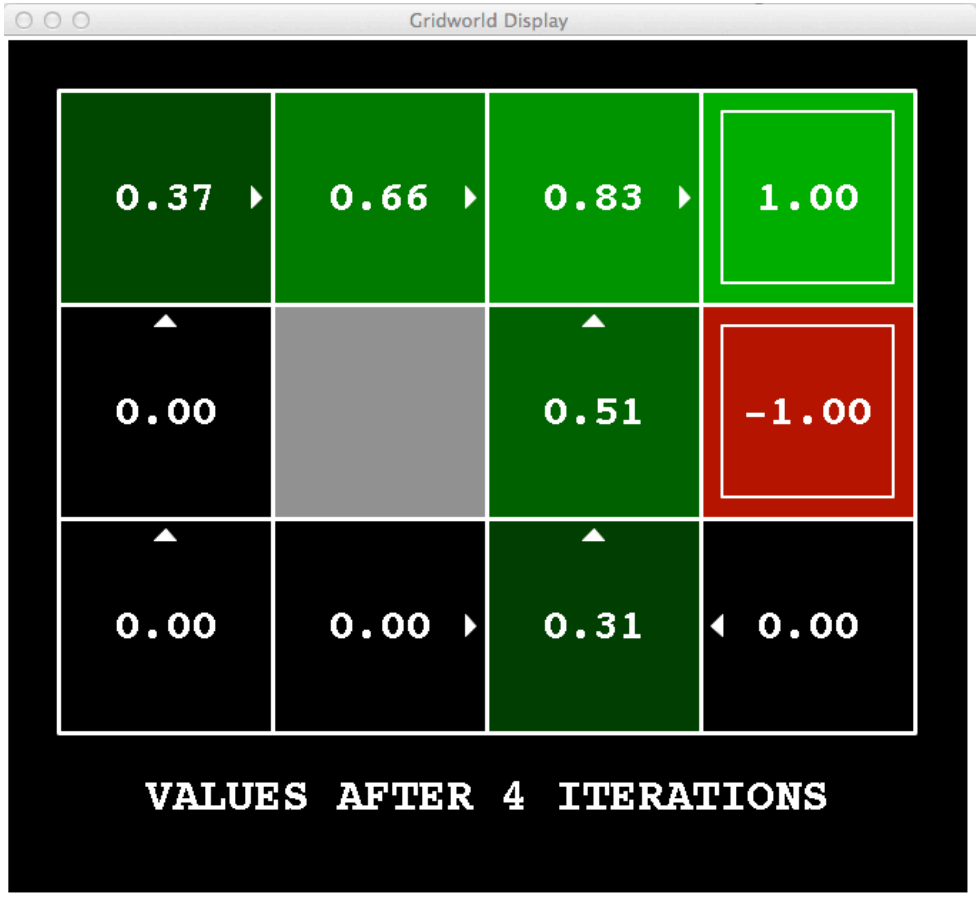
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

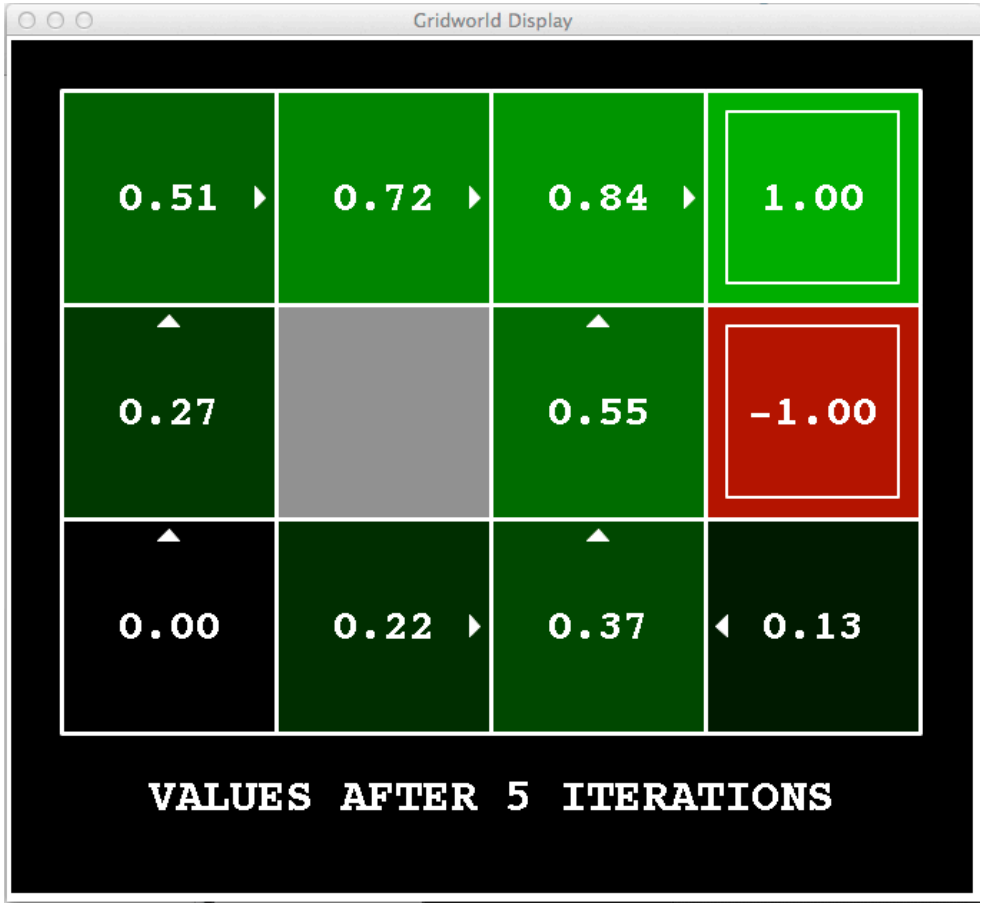
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



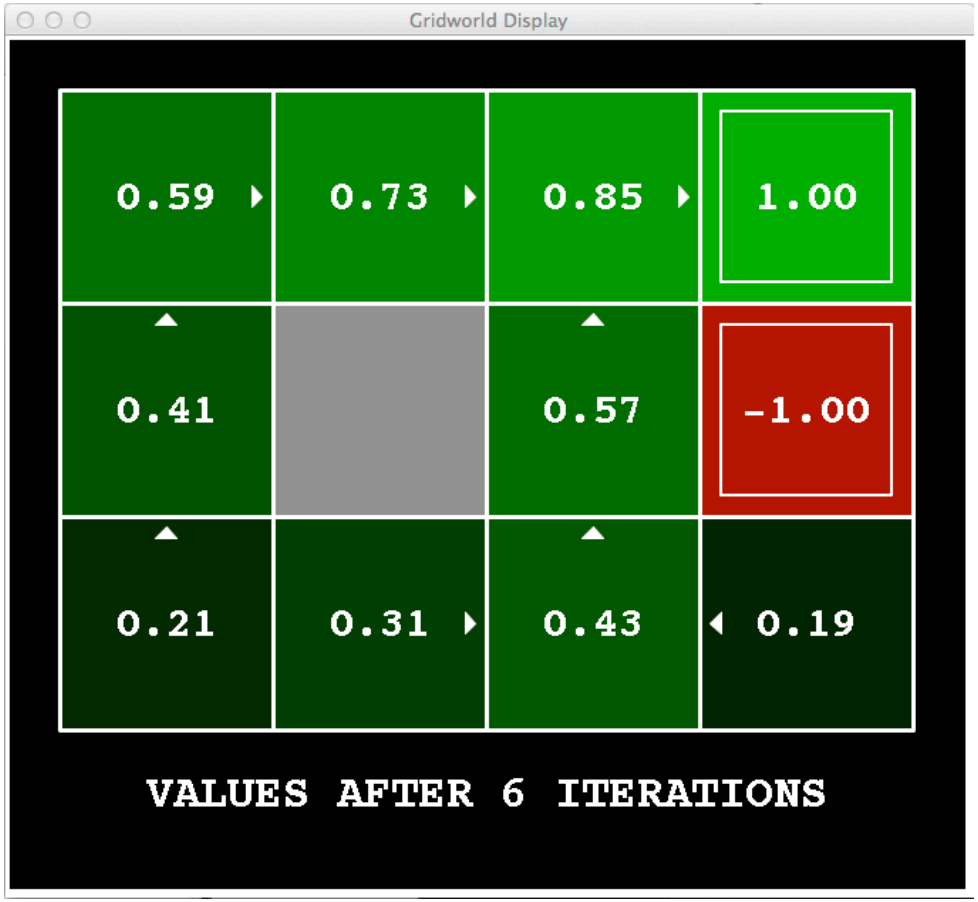
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



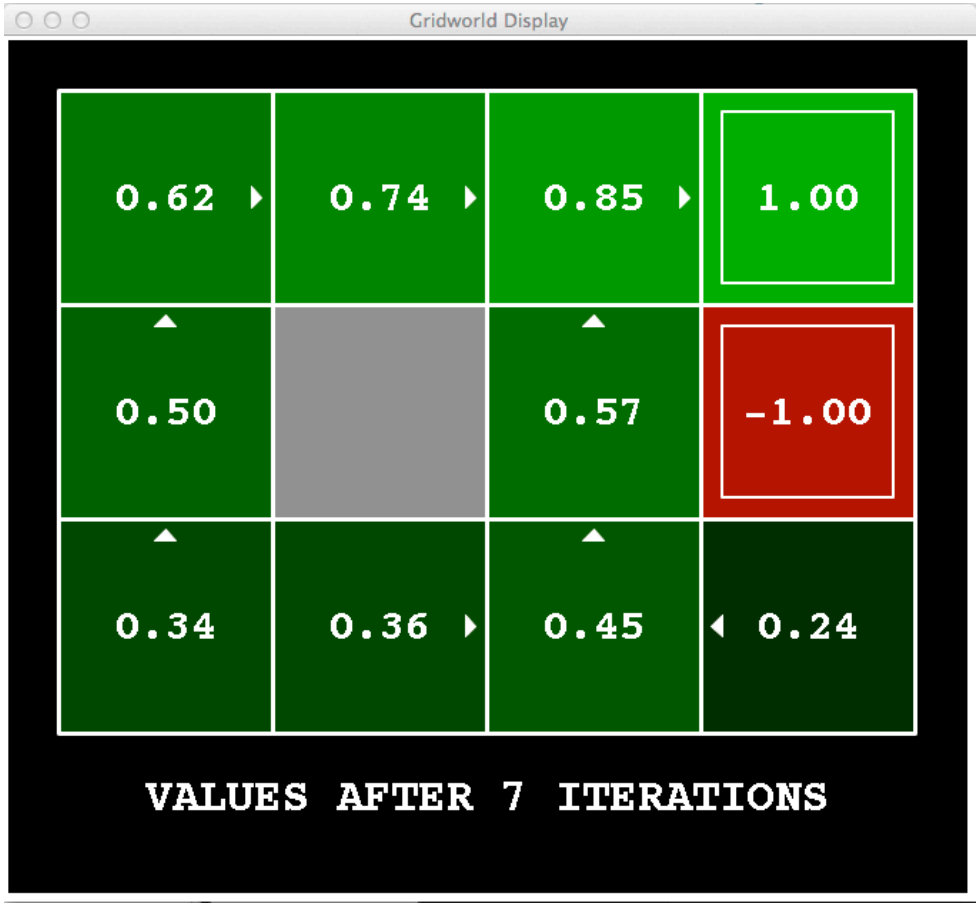
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



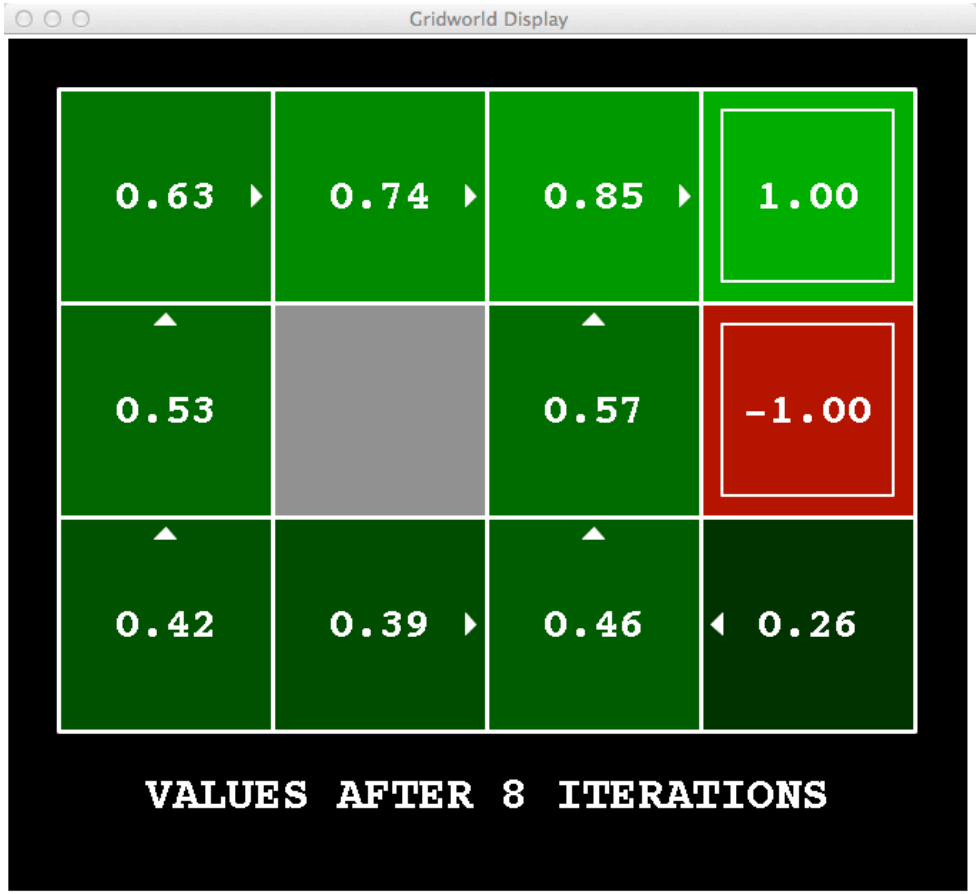
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



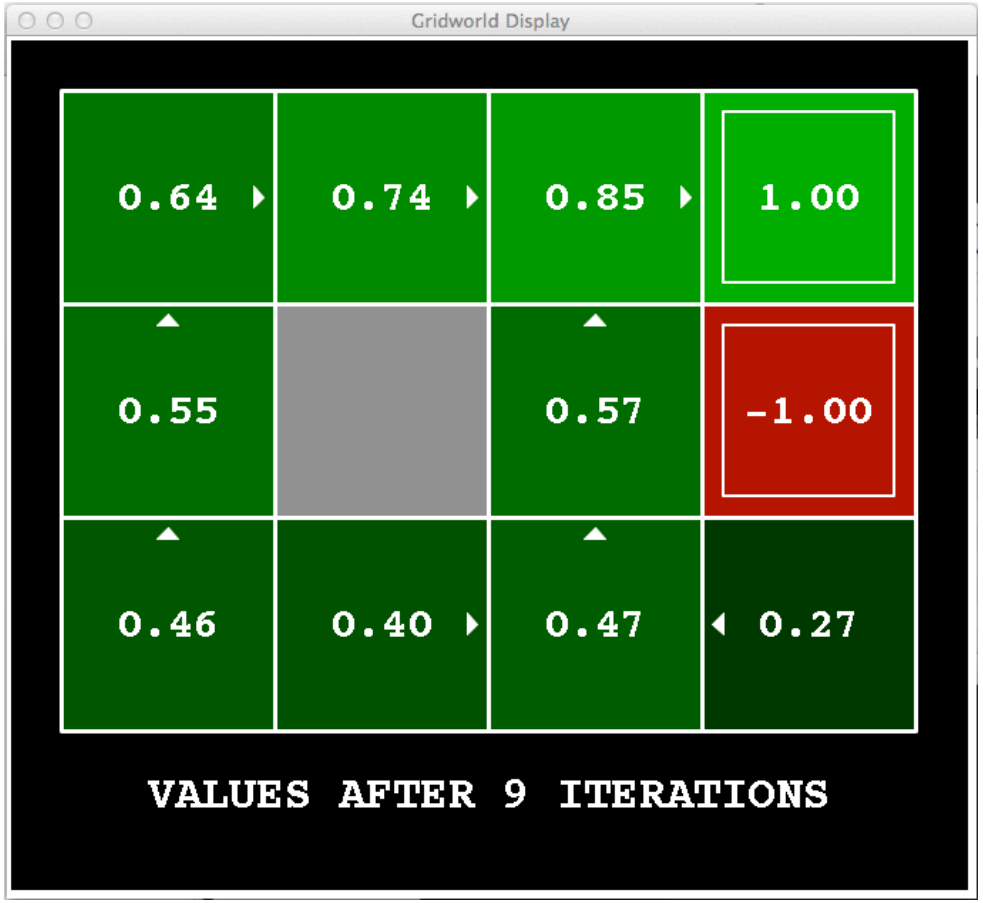
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



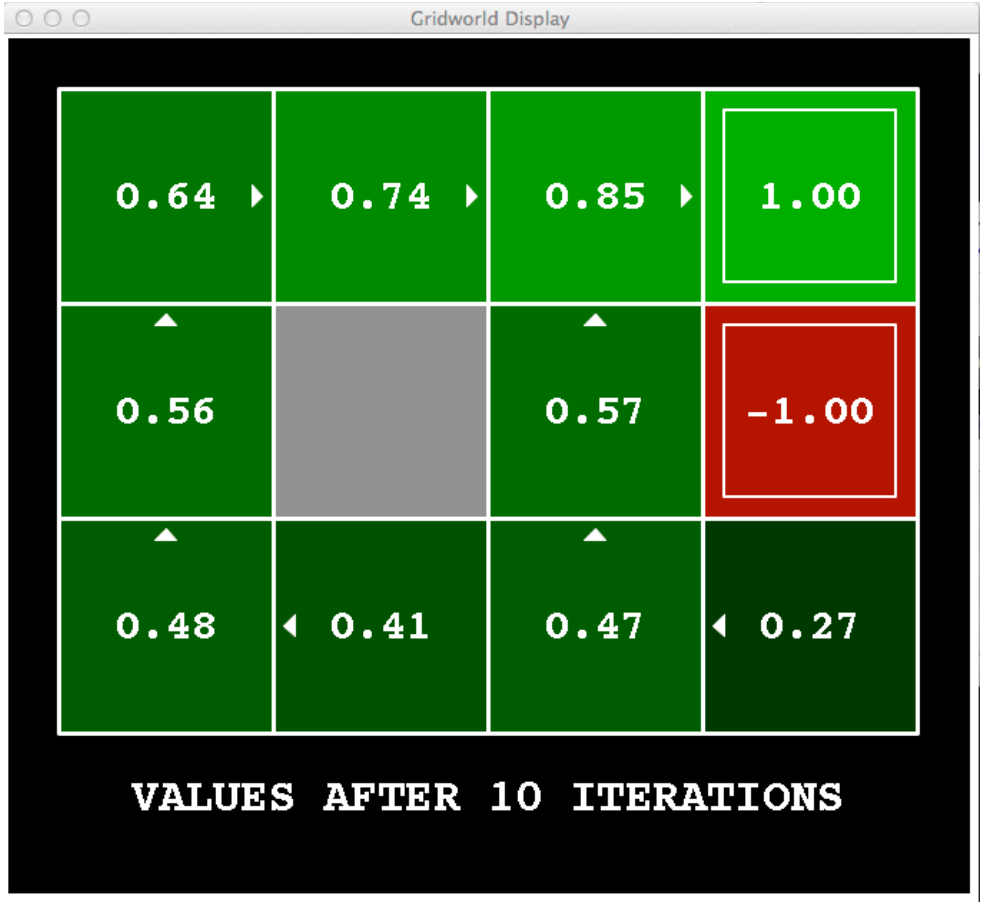
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



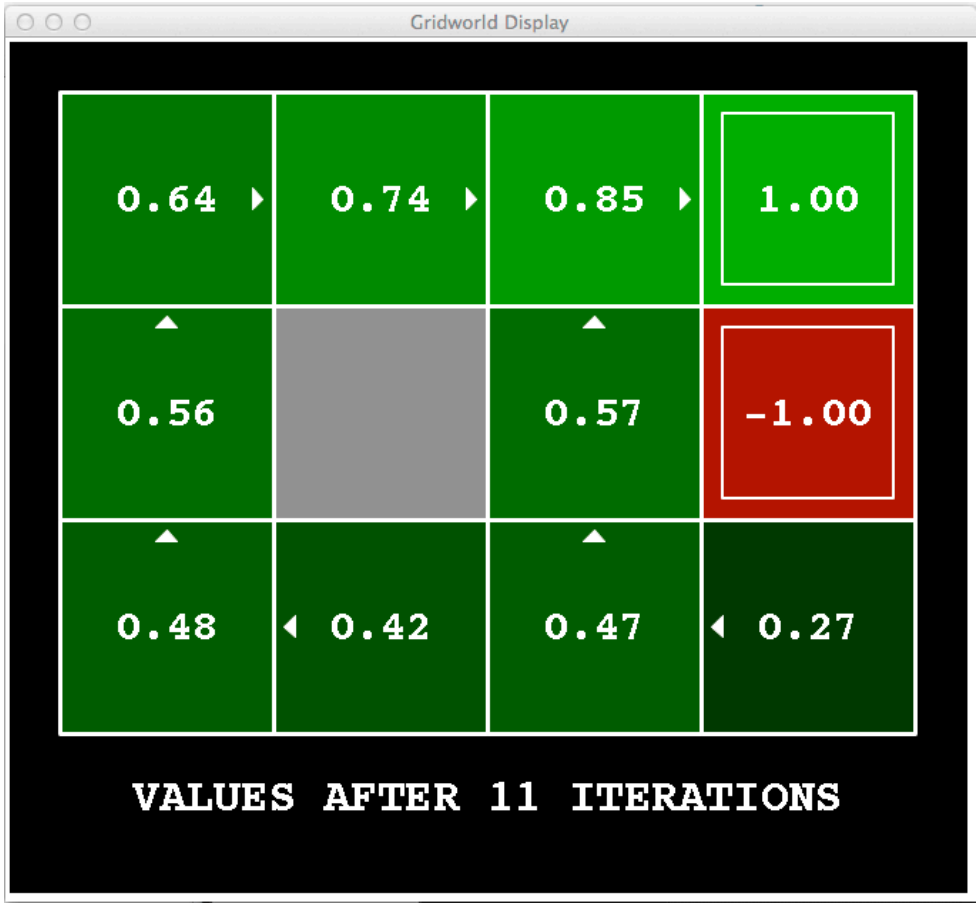
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



Noise = 0.2
Discount = 0.9
Living reward = 0

k=11

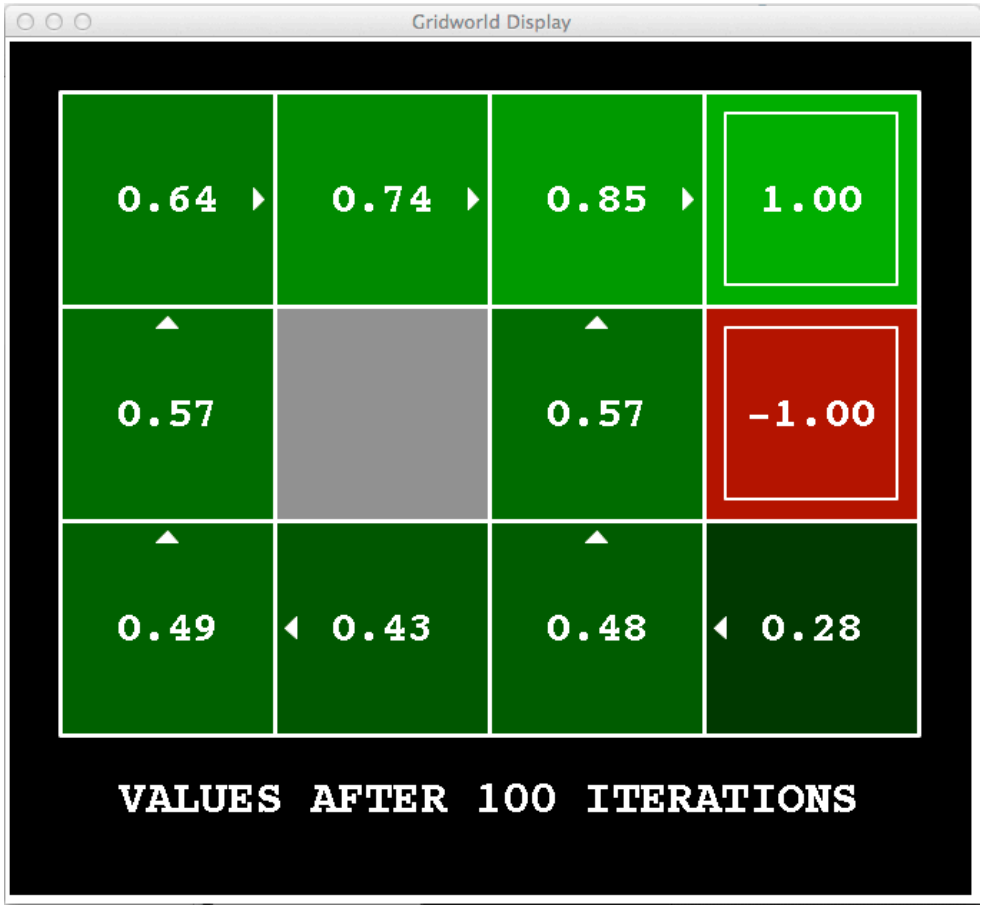


Noise = 0.2
Discount = 0.9
Living reward = 0

k=12

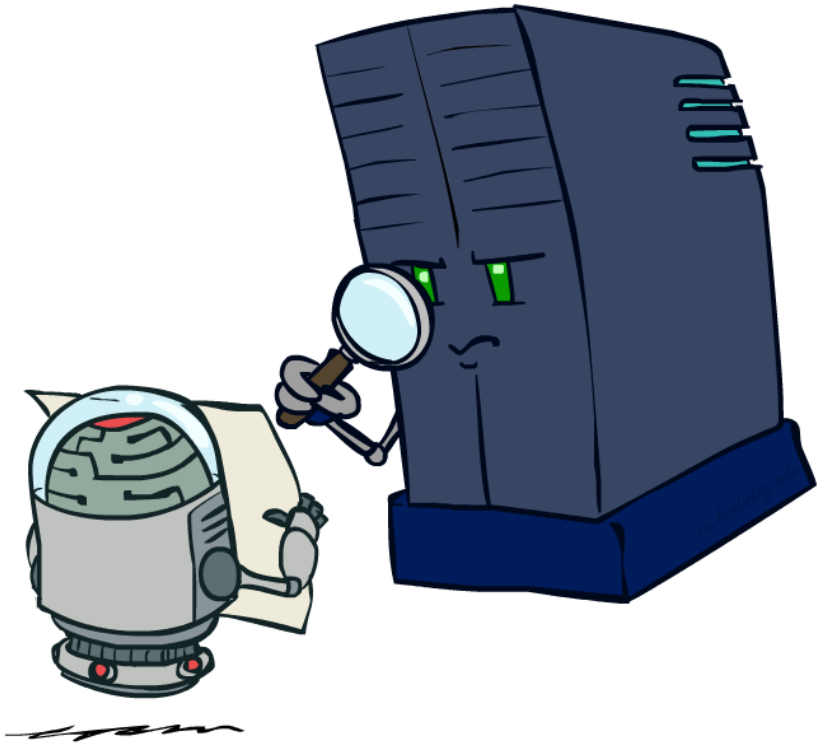
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Evaluation



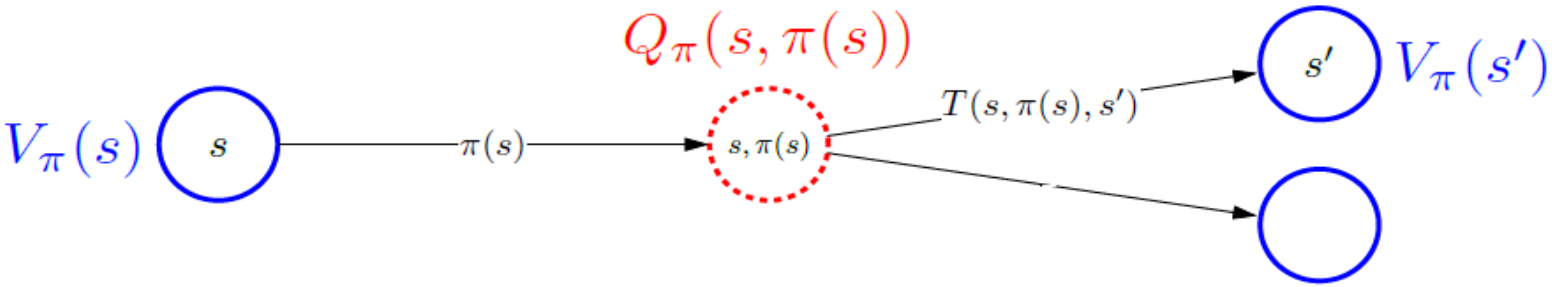
Evaluating a Policy

Definition: value of a policy

Let $V_\pi(s)$ be the expected utility received by following policy π from state s.

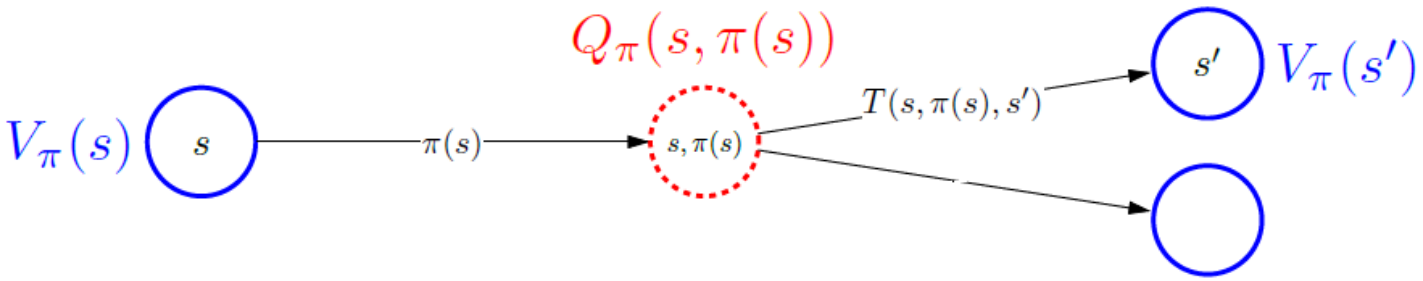
Definition: Q-value of a policy

Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s, and then following policy π.



Policy Evaluation: Approach

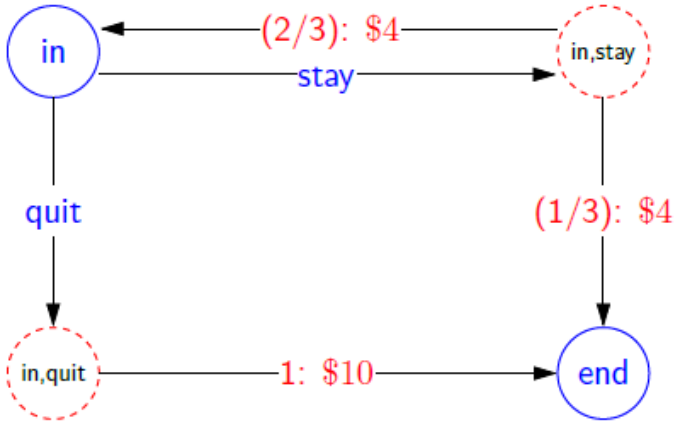
Plan: define recurrences relating value and Q-value



$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

Game 1: Policy="stay"



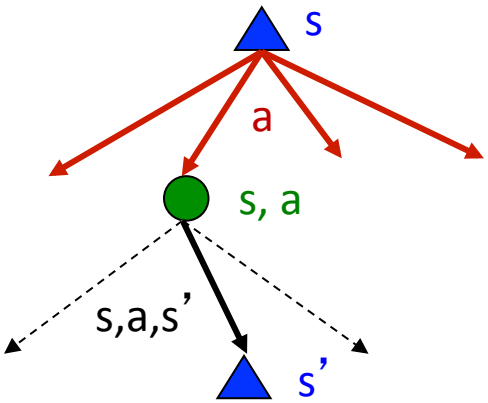
Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi(\text{in}) = Q_\pi(\text{in}, \text{stay}) = \frac{1}{3}(4) + \frac{2}{3}(4 + 1 \cdot V_\pi(\text{in}))$$

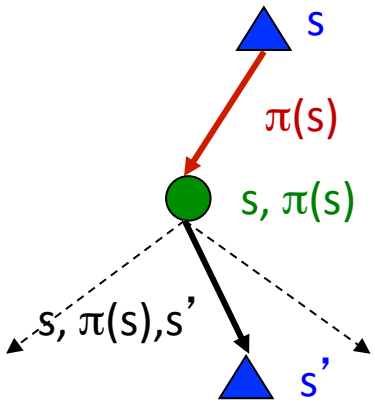
Compute in closed form: $V_\pi(\text{in}) = 12$

Fixed Policies

Do the optimal action



Do what π says to do



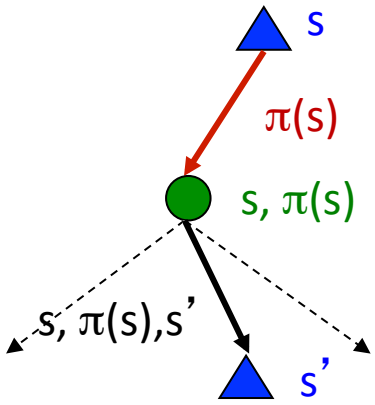
- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy π(s), then the tree would be simpler – only one action per state
  - … though the tree's value would depend on which policy we fixed

Utilities for a Fixed Policy

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

  $V^{\pi}(s)$ = expected total discounted rewards starting in s and following π
- Recursive relation (one-step look-ahead / Bellman equation):

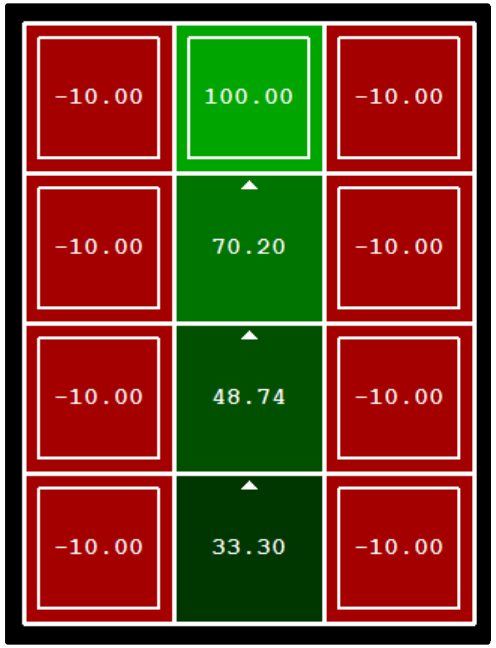
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$



Example: Policy Evaluation

π1: Always Go Right

π2: Always Go Forward



Policy Evaluation: Algorithm

Key idea: iterative improvement

Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.

Algorithm: policy evaluation

Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.

For iteration $t = 1, \ldots, t_{\text{PE}}$:

For each state s:

$$V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

Policy Evaluation: Implementation

How many iterations (t_{PE})? Repeat until values don't change much:

$$\max_s |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

Don't store $V_\pi^{(t)}$ for each iteration t, need only last two:

$$V_\pi^{(t)} \text{ and } V_\pi^{(t-1)}$$

- Some implementation notes: a good strategy for determining how many iterations to run policy evaluation is based on how accurate the result is. Rather than set some fixed number of iterations (e.g, 100), we instead set an error tolerance (e.g., $\epsilon = 0.01$), and iterate until the change between values from one iteration to the next is at most ϵ.
- The second note is that while the algorithm is stated as computing $V_\pi^{(t)}$ for each iteration t, we actually only need to keep track of the last two values.

Example: Policy Eval on Game 1

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi^{(t)}(\text{in}) = \tfrac{1}{3}(4) + \tfrac{2}{3}(4 + V_\pi^{(t-1)}(\text{in}))$$

s	end	in	
$V_\pi^{(t)}$	0.00	12.00	$(t = 100$ iterations$)$

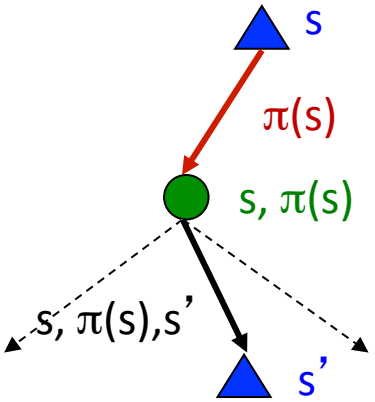
Converges to $V_\pi(\text{in}) = 12$.

Policy Evaluation

- How do we calculate the V's for a fixed policy π?
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

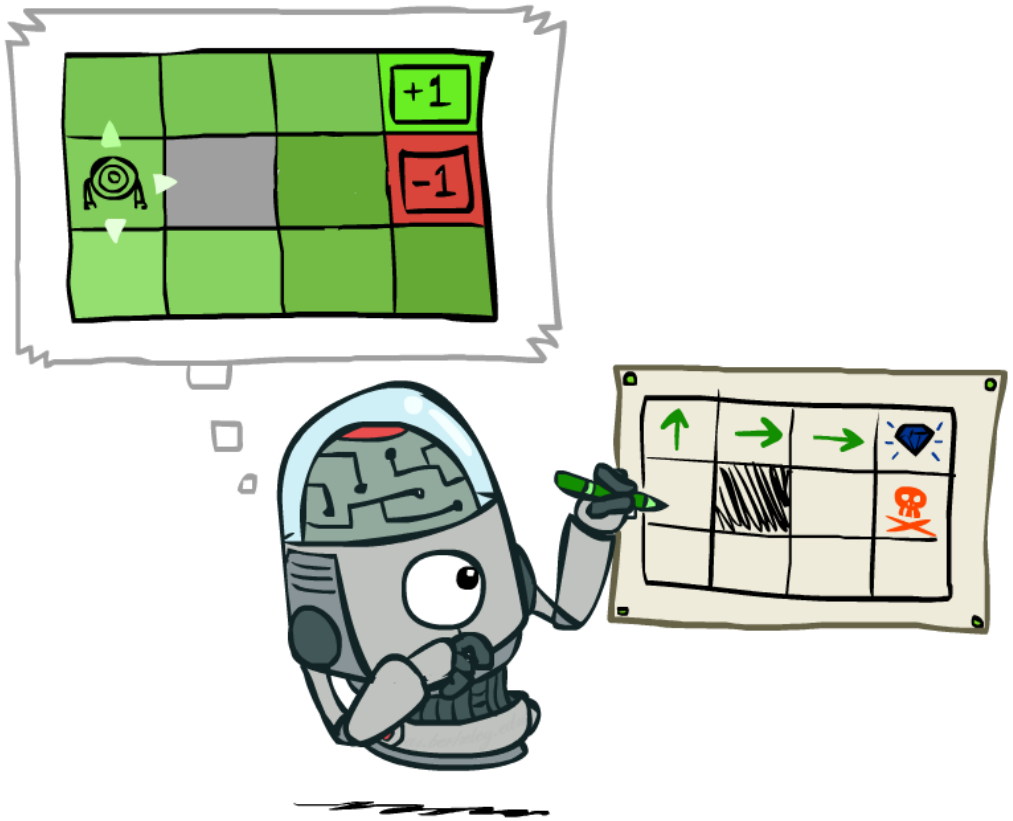
$$V_0^\pi(s) = 0$$

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



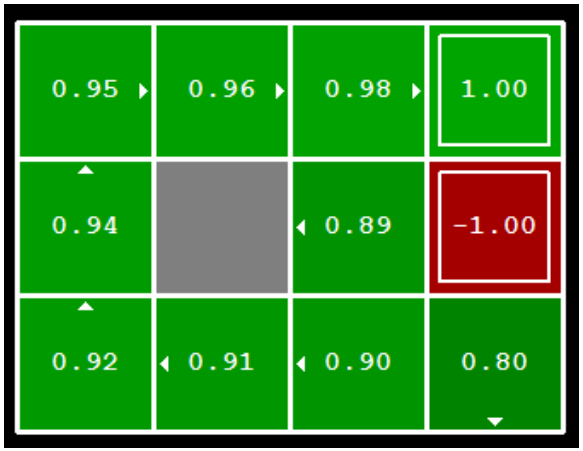
- Efficiency: $O(S^2)$ per iteration
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)

Policy 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 a mini-expectimax (one step)

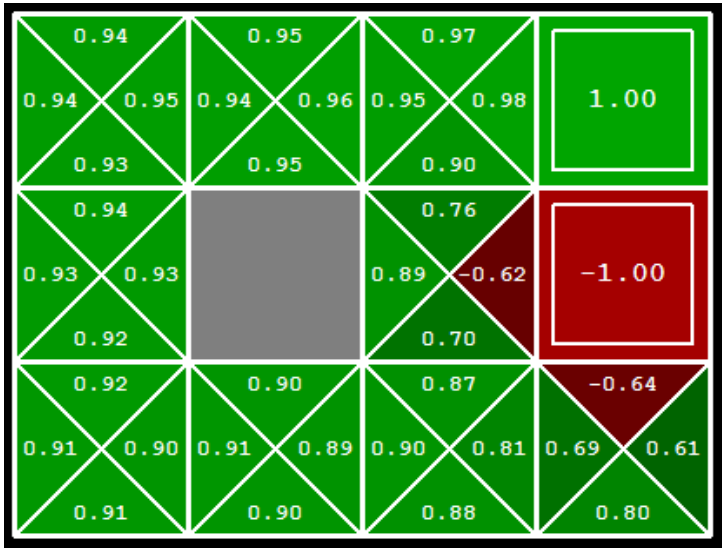


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

Computing Actions from Q-Values

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



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

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

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

Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

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 Improvement

So far: policy evaluation computes value of a fixed policy π

Next: improve π to something better π_{new}

Recall: $Q_\pi(s, a)$ is the expected utility of taking action a in state s, and then following π

Algorithm: policy improvement

Compute $Q_\pi(s, a)$ from $V_\pi(s)$

Return $\pi_{\text{new}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_\pi(s, a)$

Policy Improvement for Game 1

Suppose $\pi(\text{in}) = \text{quit}$.

Then:

$$Q_\pi(\text{in}, \text{quit}) = 10.$$

$$Q_\pi(\text{in}, \text{stay}) = 4 + \tfrac{2}{3}(10) \approxeq 10.67.$$

Policy improvement:

$$\pi_{\text{new}}(\text{in}) = \text{stay}$$

Policy Improvement: Theorem

The value of the new policy is at least that of the old one:

$$V_{\pi_{\text{new}}}(s) \geq V_{\pi}(s) \text{ for all states } s.$$

Policy Improvement: Algorithm

Algorithm: policy improvement

Compute $Q_\pi(s,a)$ from $V_\pi(s)$

Return $\pi_{\text{new}}(s) = \arg\max_{a \in \text{Actions}(s)} Q_\pi(s,a)$

MDP complexity

S states

A actions per state

S' successors (number of s' with $T(s,a,s') > 0$)

Time: $O(SAS')$

Note: unlike policy evaluation, no iterating to some error tolerance

- Policy improvement is relatively cheap. Unlike policy evaluation, we don't have to iterate. The main bottleneck is actually computing $Q_\pi(s,a)$, which takes $O(SAS')$ time. Computing the actual new policy π_{new} only takes $O(SA)$ time.

Policy Iteration

Idea: rinse and repeat

Algorithm: policy iteration

$\pi \leftarrow$ arbitrary

For $t = 1, \ldots, t_{\text{PI}}$ (or until π stops changing):

Run policy evaluation to compute V_π

Run policy improvement to get π_{new}

$\pi \leftarrow \pi_{\text{new}}$

Time: $O(t_{\text{PI}}(t_{\text{PE}}SS' + SAS'))$

Implementation trick: warm start policy evaluation with previous V_π

Comparison

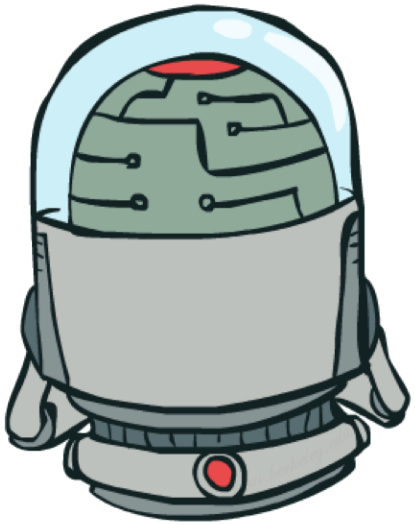
- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do 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)
- Both are dynamic programs for solving MDPs

Summary: MDP Algorithms

- So you want to….
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

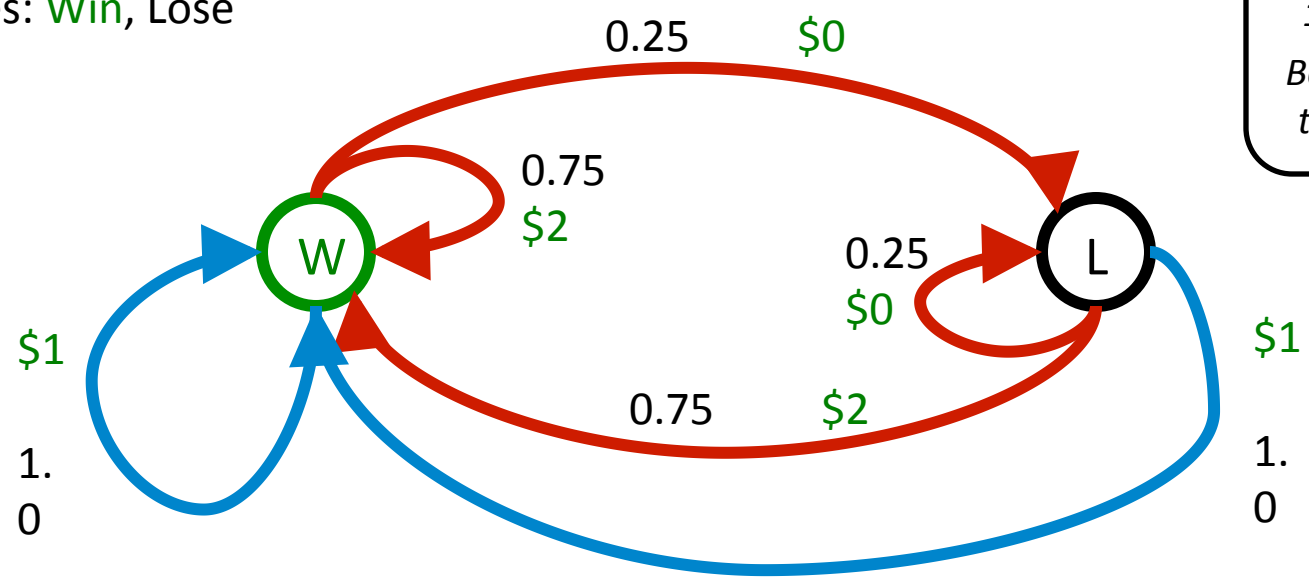
- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions

Double Bandits



Double-Bandit MDP

- Actions: *Blue*, *Red*
- States: Win, Lose



No discount
100 time steps
Both states have the same value

Offline Planning

- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

No discount
100 time steps
Both states have the same value

	Value
Play Red	150
Play Blue	100

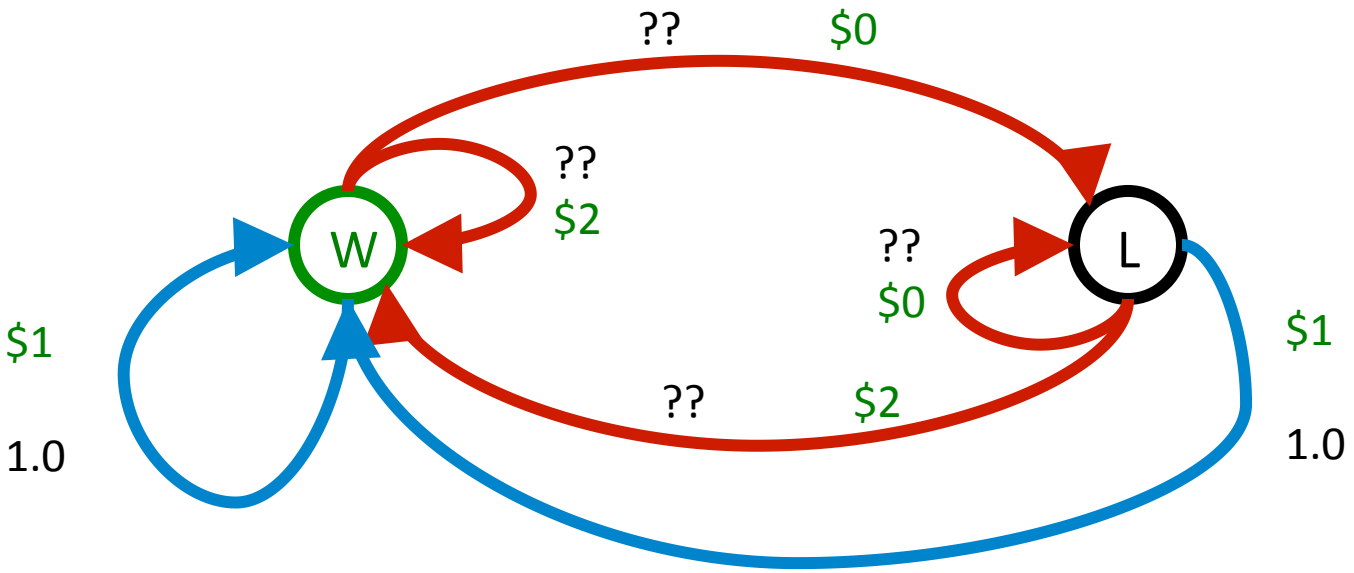
Let's Play!

$2 $2 $0 $2 $2

$2 $2 $0 $0 $0

Online Planning

- Rules changed! Red's win chance is different.



Let's Play!

$0 $0 $0 $2 $0

$2 $0 $0 $0 $0

What Just Happened?

- That wasn't planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn't solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP

Next Time: Reinforcement Learning!