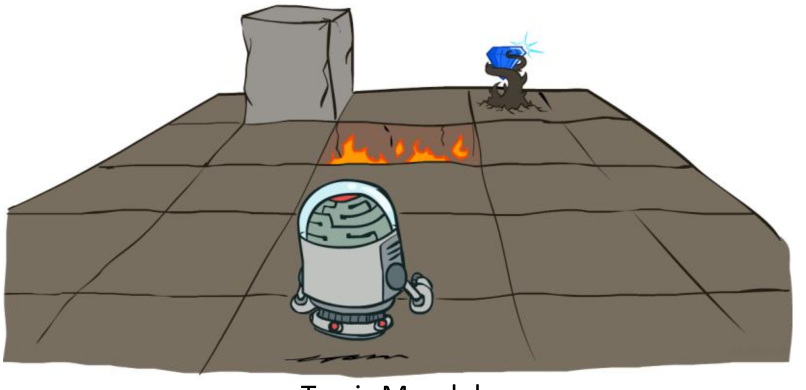
CS 473: Artificial Intelligence

MDP Planning: Value Iteration and Policy Iteration



Travis Mandel *(subbing for Dan Weld)*

University of Washington

Slides by Dan Klein & Pieter Abbeel / UC Berkeley. (http://ai.berkeley.edu) and by Dan Weld, Mausam & Andrey Kolobov

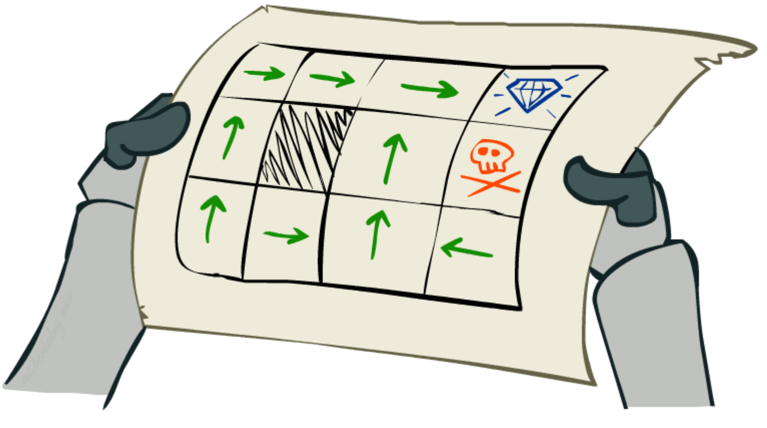
Reminder: Midterm Monday!!

- Will cover everything from Search to Value Iteration
  - One page (double-sided, 8.5 x 11) notes allowed

Reminder: MDP Planning

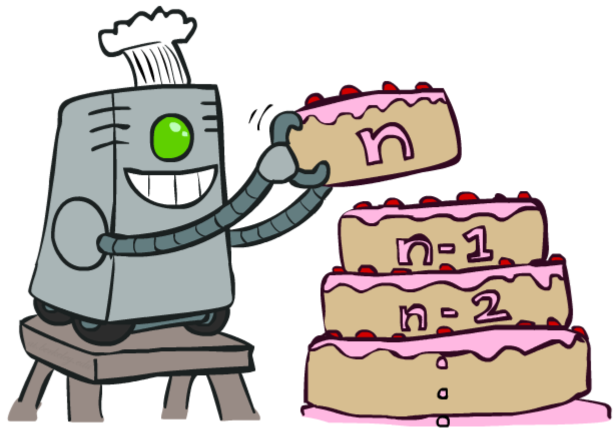
- Given an MDP, find optimal policy π*: S→A that maximizes expected discounted reward
  - Sometimes called “Solving” the MDP
- Being so long-term complicates things
  - Simplifies things if we know long-term value of state

MDP Planning



- Value Iteration
  - Prioritized Sweeping
- Policy Iteration

Value Iteration



Value Iteration

Called a "Bellman Backup"

- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat**

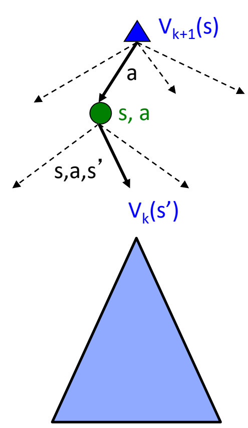
  K += 1

  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

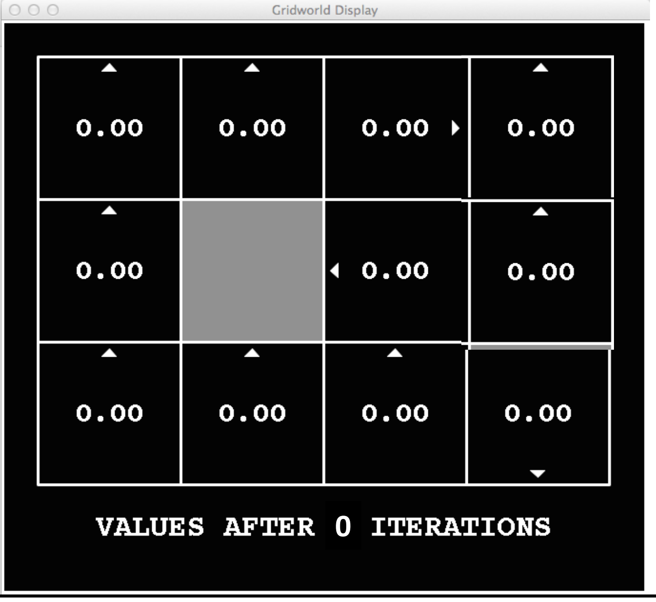
  } do ∀s, a

- **Repeat until $|V_{k+1}(s) – V_k(s)| < \varepsilon$, forall s** *"convergence"*



Successive approximation; dynamic programming

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1

If agent is in 4,3, it only has one legal action: get jewel. It gets a reward and the game is over.
If agent is in the pit, it has only one legal action, die. It gets a penalty and the game is over.

Agent does NOT get a reward for moving INTO 4,3.

Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

0.8 (0 + 0.9*1)
+ 0.1 (0 + 0.9*0)
+ 0.1 (0 + 0.9*0)

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

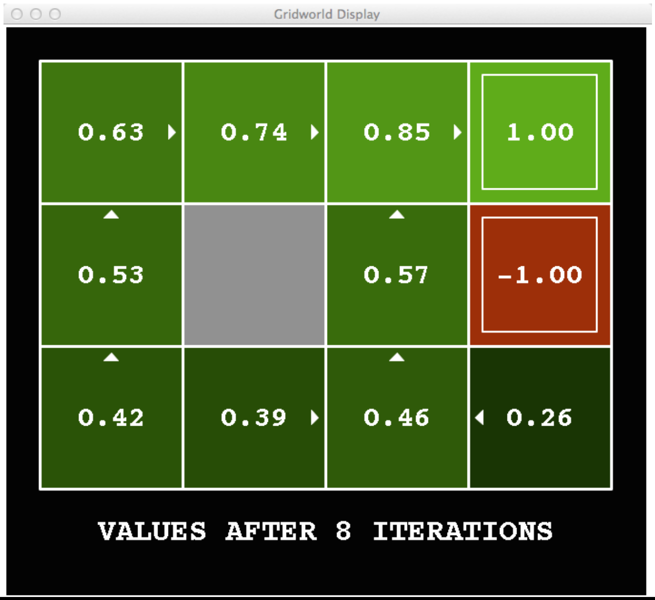
k=6

Noise = 0.2
Discount = 0.9
Living reward = 0

k=7

Noise = 0.2
Discount = 0.9
Living reward = 0

k=8

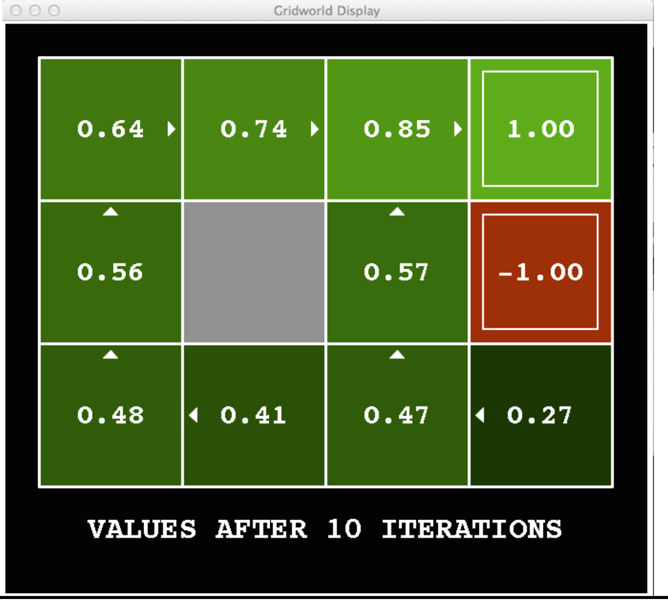


Noise = 0.2
Discount = 0.9
Living reward = 0

k=9

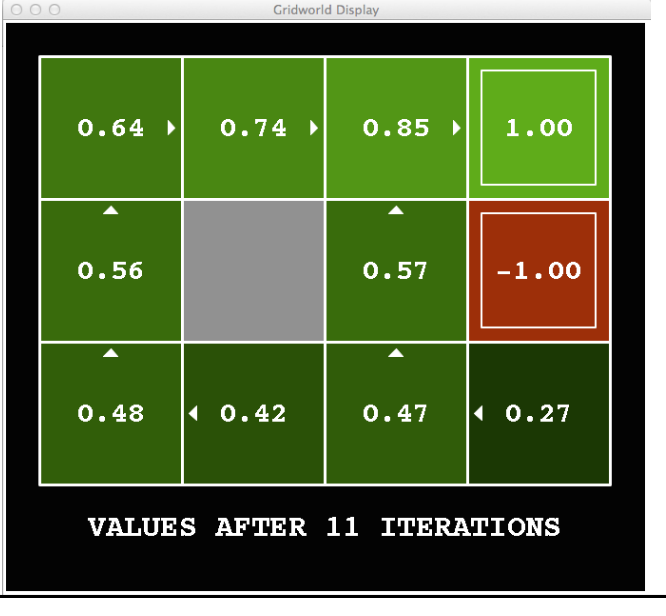
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



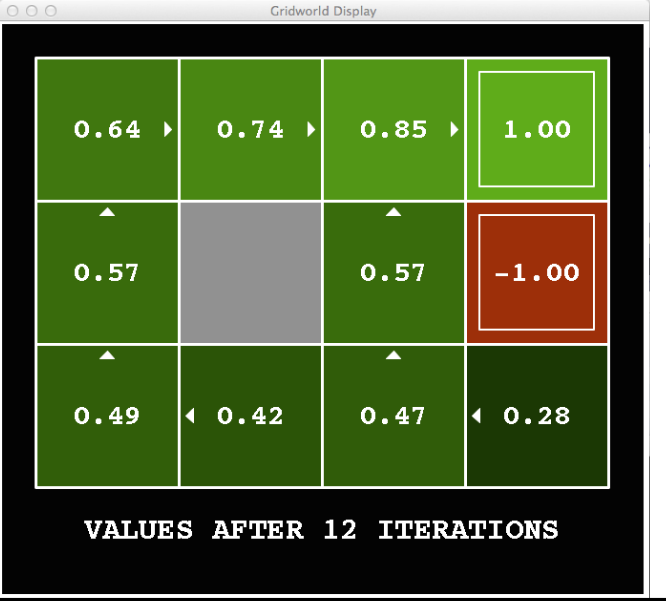
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



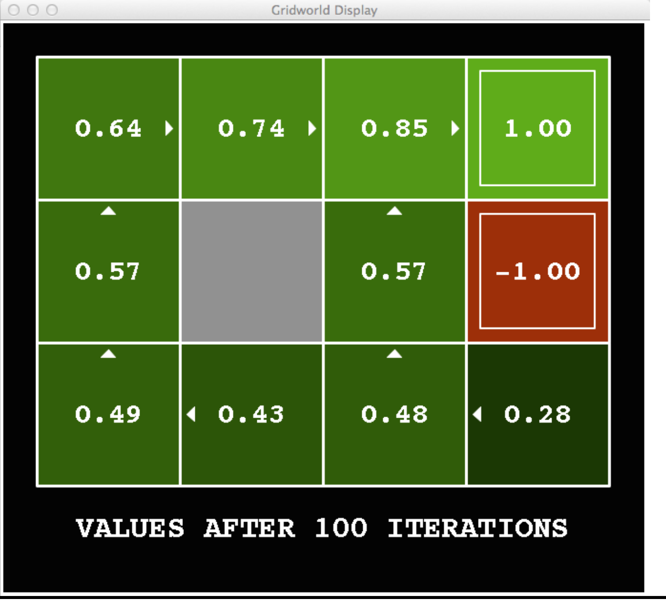
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



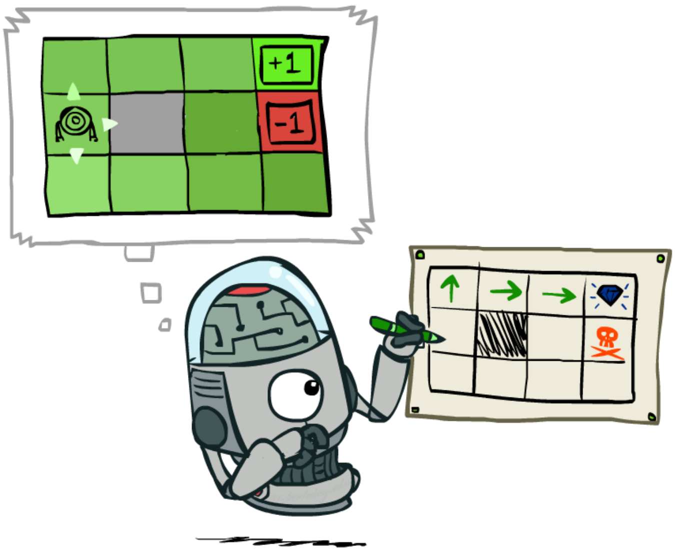
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



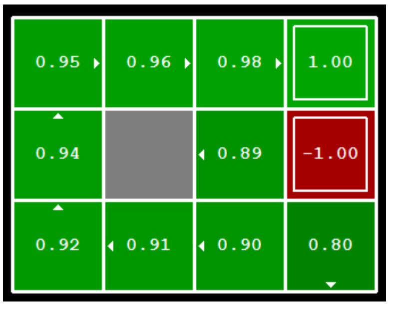
Noise = 0.2
Discount = 0.9
Living reward = 0

VI: Policy Extraction



Computing Actions from Values

- Let's imagine we have the optimal values V*(s)
- How should we act?
  - In general, it's not obvious!
- We need to do a mini-expectimax (one step)

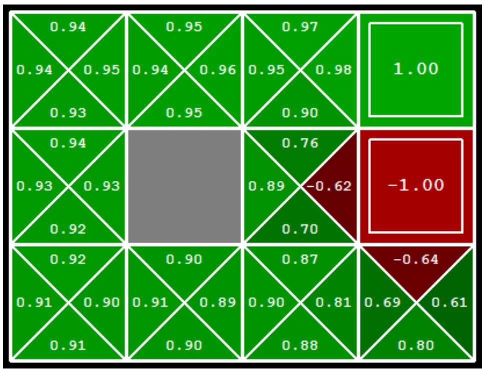


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- 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:



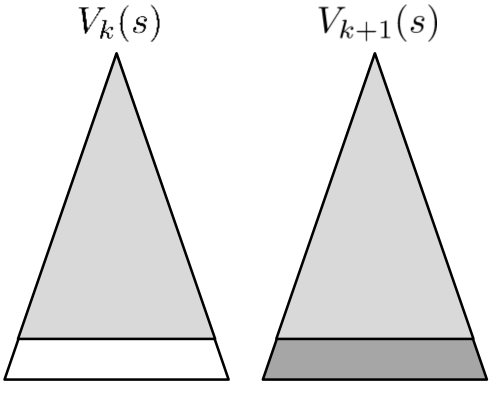
- How should we act?
  - Completely trivial to decide!

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

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

Convergence*

- How do we know the V_k vectors will 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 max difference happens if big reward at k+1 level
  - That last layer is at best all R_{MAX}
  - 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



Value Iteration - Recap

- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat** *do Bellman backups*

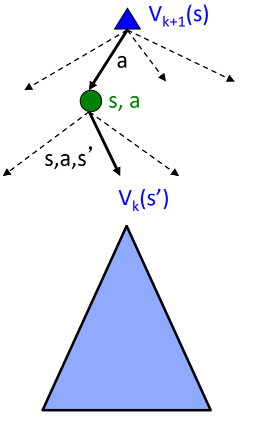
  K += 1

  Repeat for all states, s, and all actions, a:

  $$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

  $$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$$

- **Until $|V_{k+1}(s) - V_k(s)| < \varepsilon$, forall s** *"convergence"*
- **Theorem: will converge to unique optimal values**



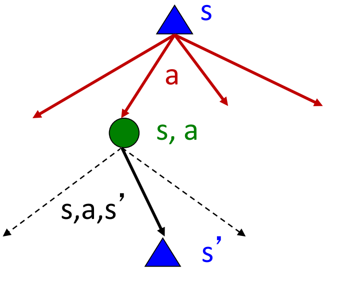
Problems with Value Iteration

- Value iteration repeats the Bellman updates:

  $$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

  $$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$$

- 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)]

VI → Asynchronous VI

- Is it essential to back up ***all*** states in each iteration?
  - No!
- States may be backed up
  - many times or not at all
  - in any order
- As long as no state gets starved…
  - convergence properties still hold!!

30

k=1

Noise = 0.2
Discount = 0.9
Living reward = 0

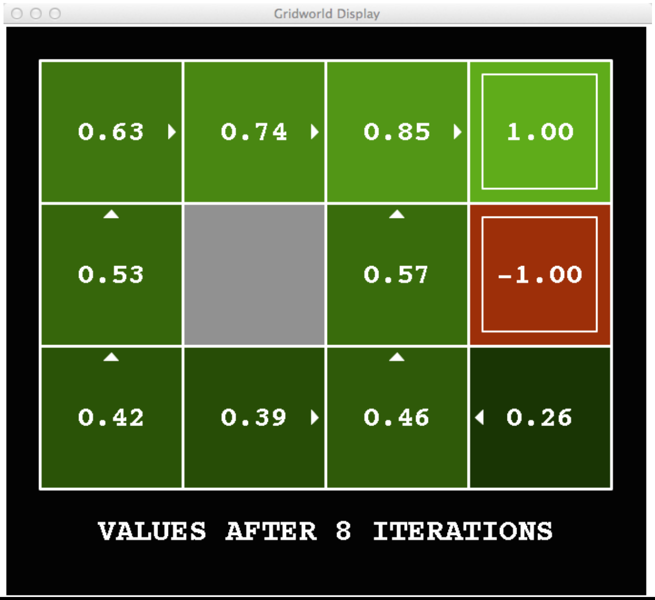
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

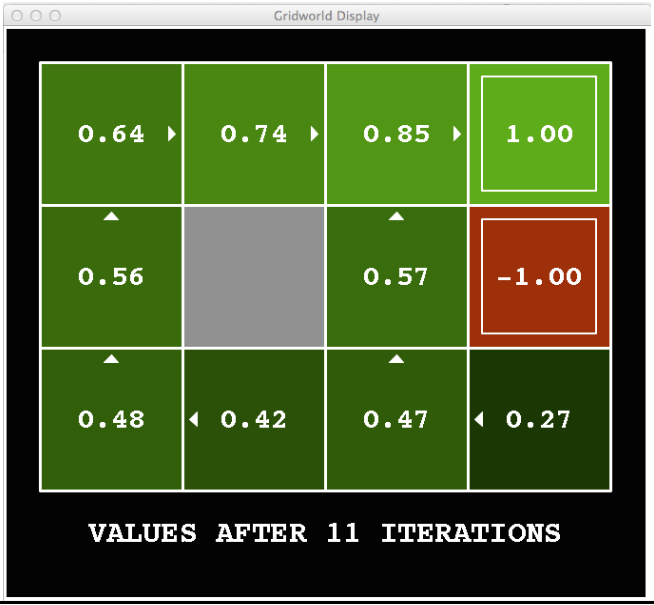
k=9

Noise = 0.2
Discount = 0.9
Living reward = 0

k=10

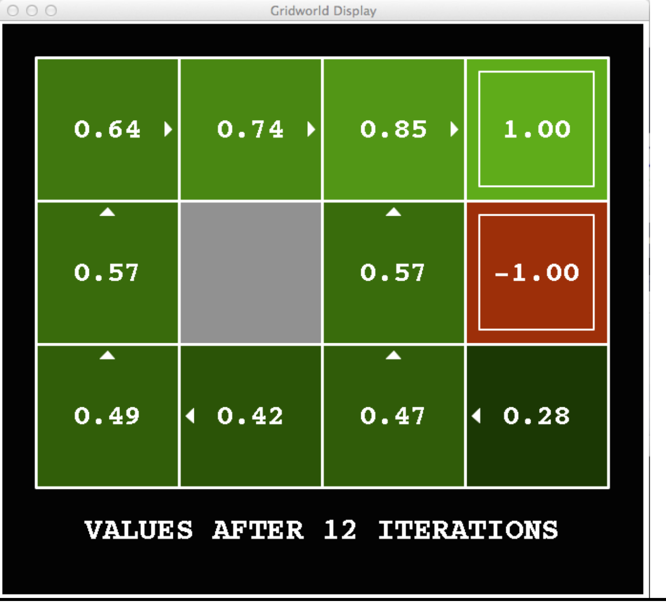
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



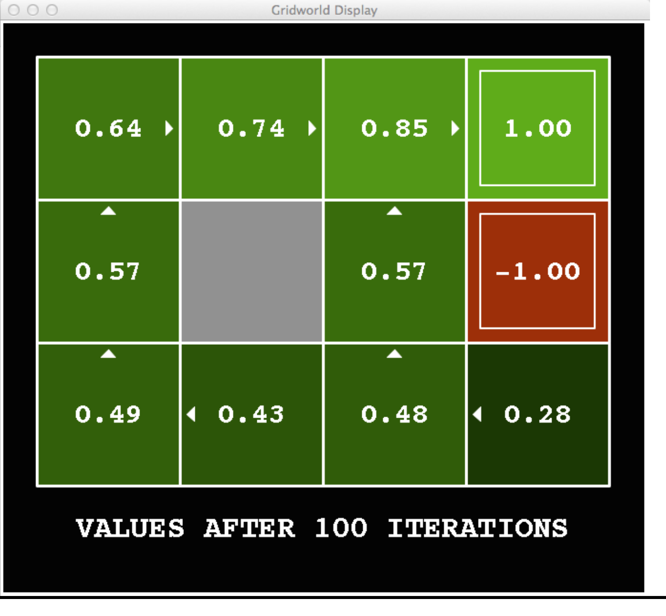
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

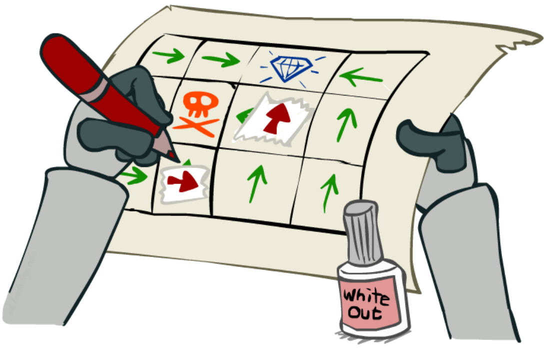
Asynch VI: Prioritized Sweeping

- Why backup a state if values of successors ***unchanged***?
- Prefer backing a state
  - whose successors had ***most*** change
- Priority Queue of (state, expected change in value)
- Backup in the order of priority
- After backing up state s', update priority queue
  - for all predecessors s (ie all states where an action can reach s')
  - Priority(s) $\leftarrow$ T(s,a,s') * $|V^{k+1}(s') - V^{k}(s')|$

Prioritized Sweeping

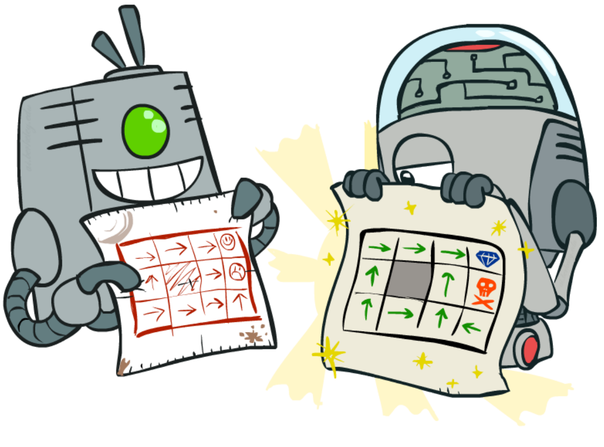
- Pros?
- Cons?

MDP Planning



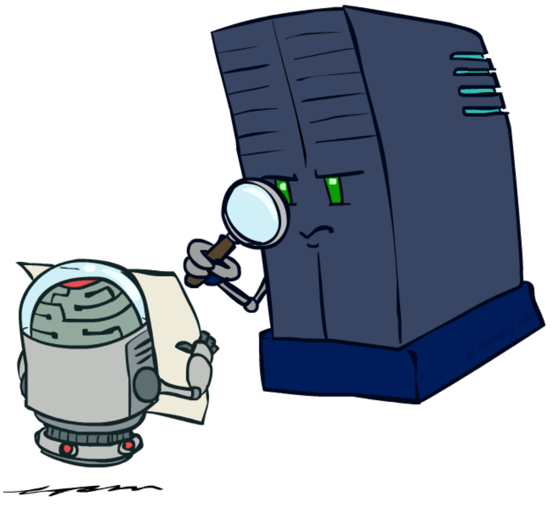
- Value Iteration
  - Prioritized Sweeping
- Policy Iteration

Policy Methods

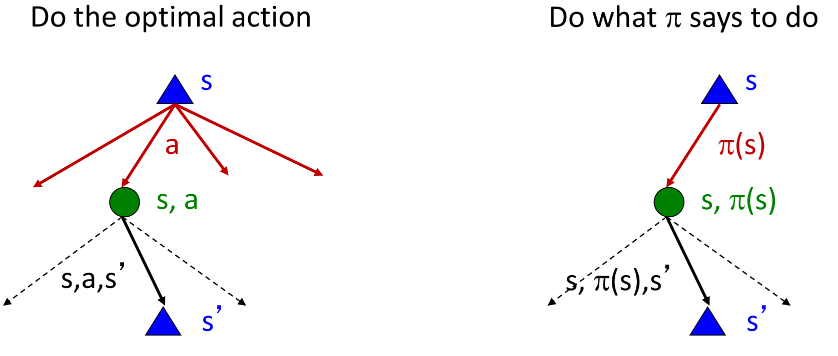


Policy Iteration =
1. Policy Evaluation
2. Policy Improvement

Part 1 - Policy Evaluation



Fixed Policies



- 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
  - … though the tree's value would depend on which policy we fixed

Computing Utilities for a Fixed Policy

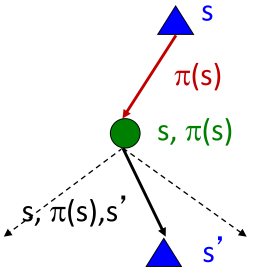
- A new basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- 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

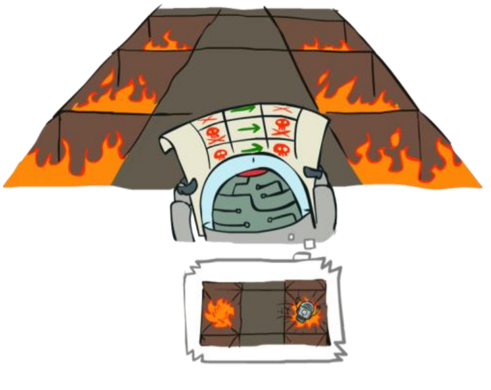
- Recursive relation (variation of Bellman equation):

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

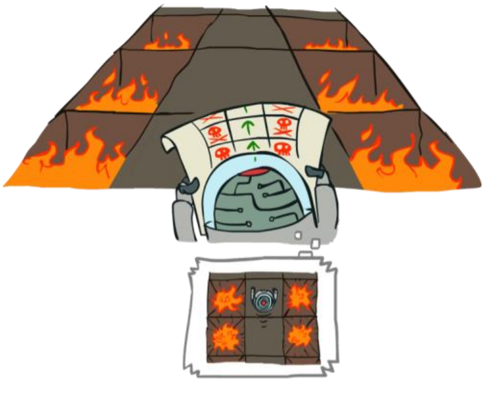


Example: Policy Evaluation

Always Go Right



Always Go Forward



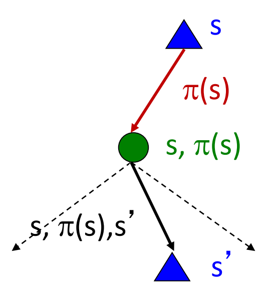
Example: Policy Evaluation

Always Go Right

Always Go Forward

Iterative Policy Evaluation Algorithm

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



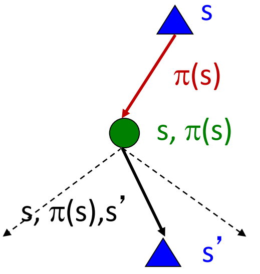
$$V_0^\pi(s) = 0$$

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

- Efficiency: O(S^2) per iteration
  - Often converges in much smaller number of iterations compared to VI

Linear Policy Evaluation Algorithm

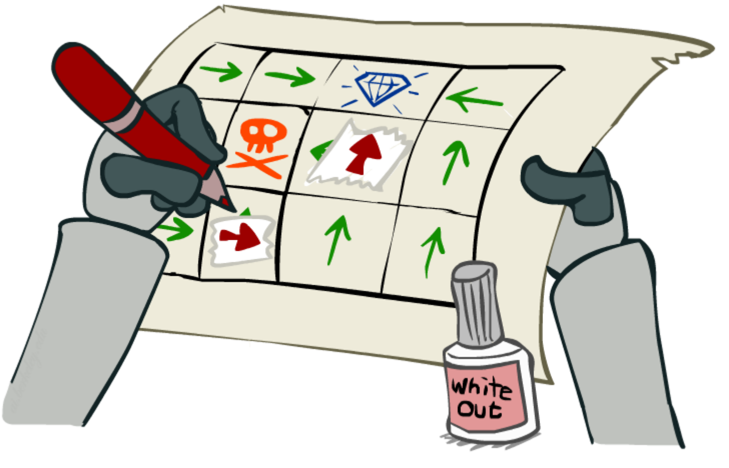
- How do we calculate the V's for a fixed policy π?
- Idea 2: Without the maxes, the Bellman equations are just a linear system of equations



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

- Solve with Matlab (or your favorite linear system solver)
  - S equations, S unknowns = $O(S^3)$ and EXACT!
  - In large spaces, still too expensive

Part 2 - Policy Iteration



Policy Iteration

- Initialize π(s) to random actions
- Repeat
  - Step 1: Policy evaluation: calculate utilities of π at each s using a nested loop
  - Step 2: Policy improvement: update policy using one-step look-ahead

    "For each s, what's the best action I could execute, assuming I then follow π?

    Let π'(s) = this best action.

    π = π'
- Until policy doesn't change

Policy Iteration Details

- Let i =0
- Initialize $\pi_i(s)$ to random actions
- Repeat
  - Step 1: Policy evaluation:
    - Initialize k=0; Forall s, $V_0^\pi(s) = 0$
    - Repeat until V^π converges
      - For each state s, $V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')\right]$
      - Let k += 1
  - Step 2: Policy improvement:
    - For each state, s, $\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s')\right]$
    - If $\pi_i == \pi_{i+1}$ then it's optimal; return it.
    - Else let i += 1

Example

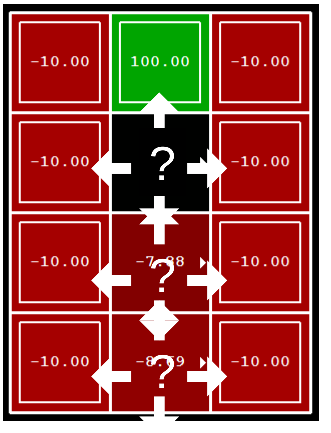
Initialize π_0 to "always go right"

Perform policy evaluation

Perform policy improvement
    Iterate through states

Has policy changed?

Yes! i += 1



Example

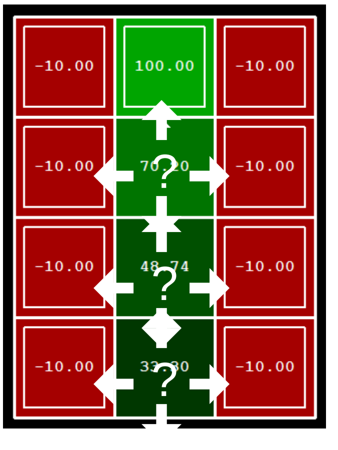
π_1 says "always go up"

Perform policy evaluation

Perform policy improvement
    Iterate through states

Has policy changed?

No! We have the optimal policy



Example: Policy Evaluation

Always Go Right

Always Go Forward

Policy Iteration Properties

- Policy iteration finds the optimal policy, guaranteed (assuming exact evaluation)!
- Often converges (much) faster

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
  - What is the space being searched?
- In policy iteration:
  - We do fewer iterations
  - Each one is slower (must update all V^{π} and then choose new best π)
  - What is the space being searched?
- Both are dynamic programs for planning in MDPs