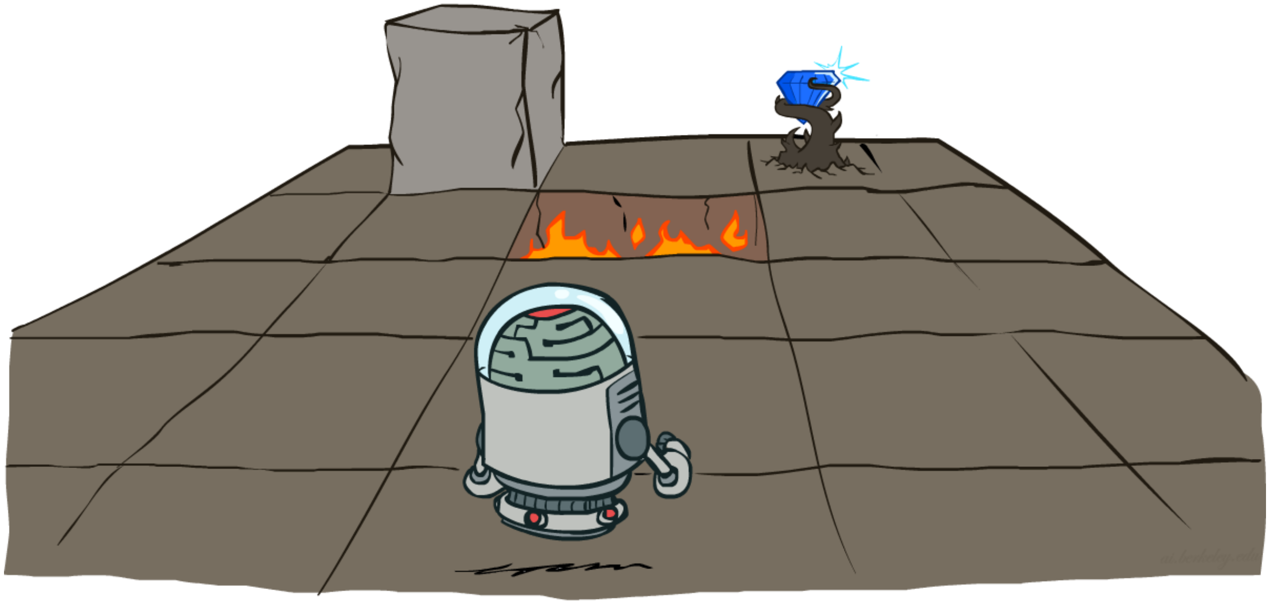
CS 573: Artificial Intelligence

Markov Decision Processes



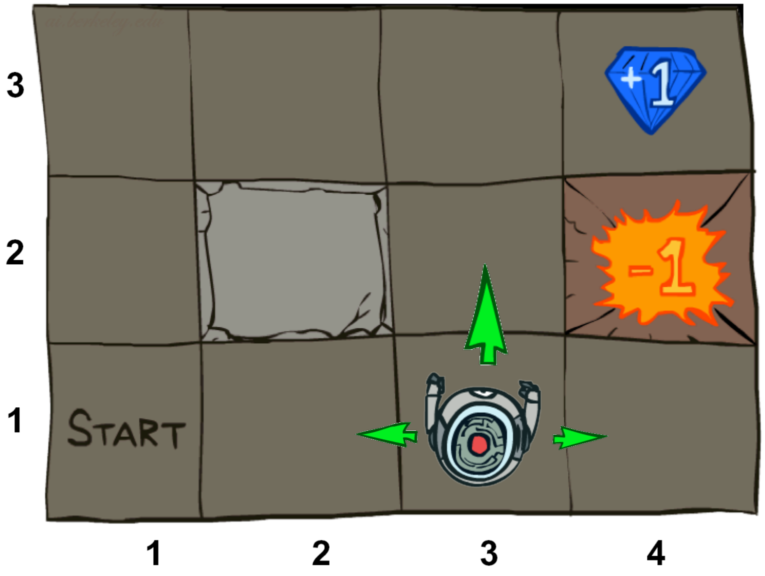
Dan Weld

University of Washington

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

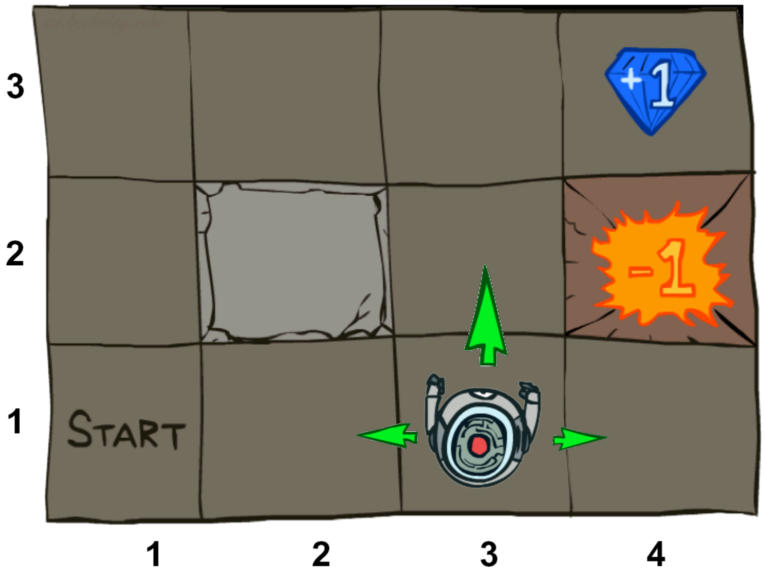
Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent’s path
- Noisy movement: 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)
  - Big rewards come at the end (good or bad)
- Goal: ~ maximize sum of rewards



Markov Decision Processes

- An MDP is defined by:
  - A set of states s ∈ S
  - A set of actions a ∈ 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'), e.g. in R&N
  - A start state
  - Maybe a terminal state



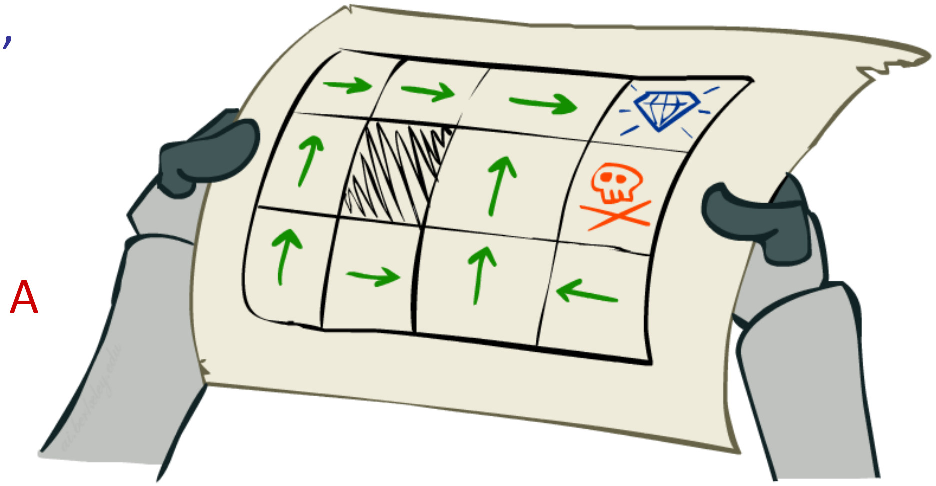
- MDPs are non-deterministic search problems
  - One way to solve them is with expectimax search
  - We'll have a new tool soon

Input: MDP Output: Policy

- 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

- Expectimax didn't output an entire policy
  - It computed the action for a single state only



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

Example: Racing

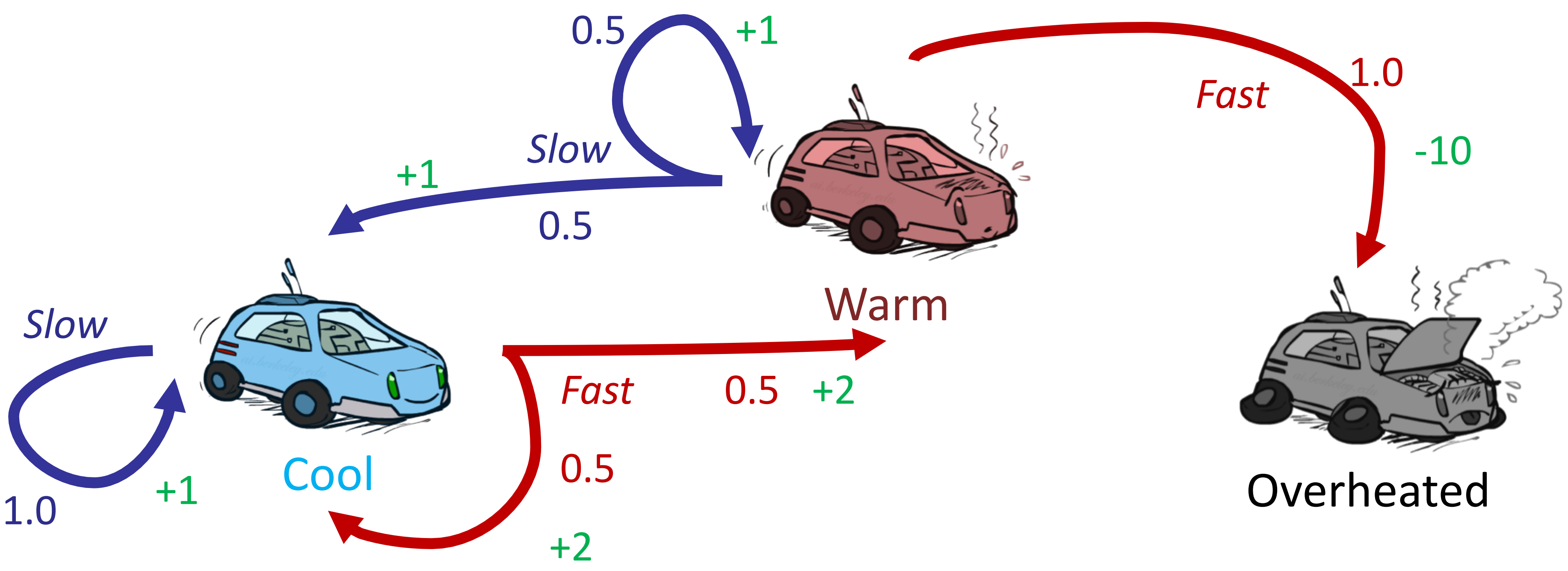
S ?

A ?

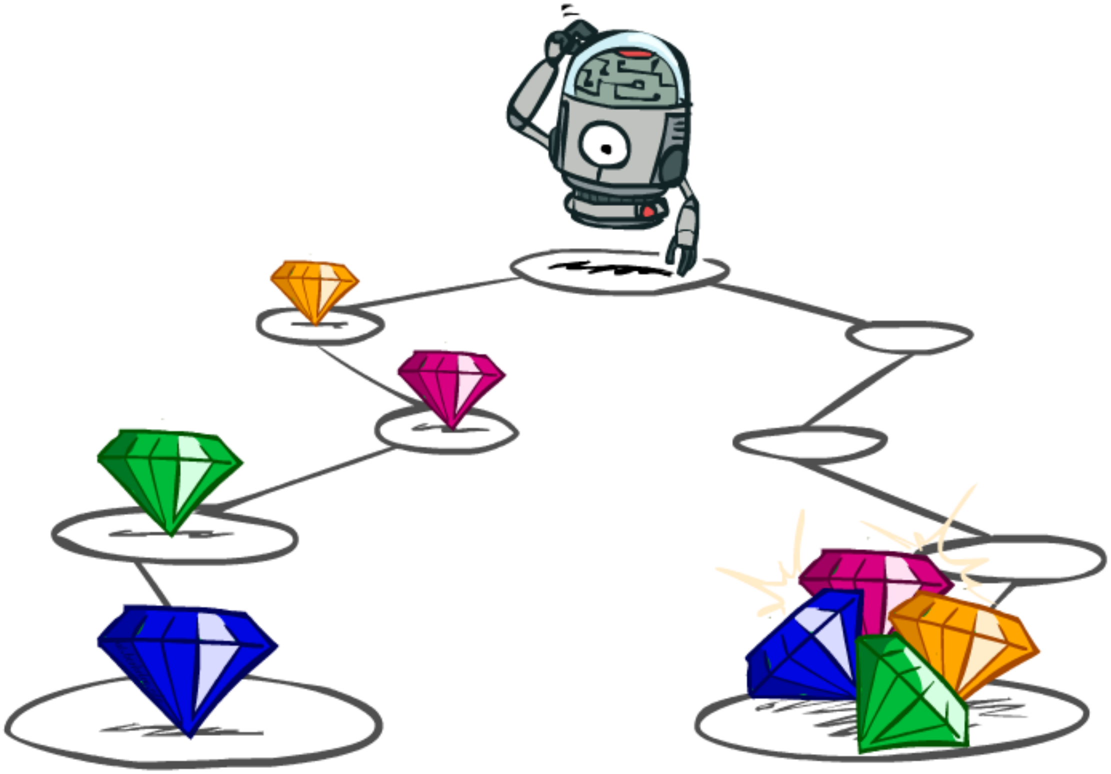
T ?

R ?

S_0 ?

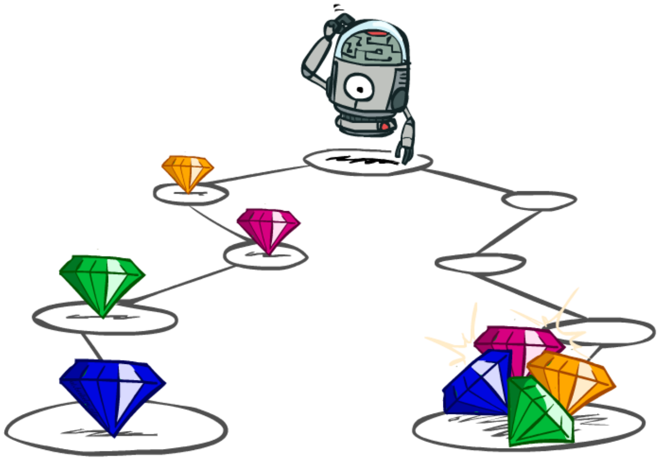


Utilities of Sequences



Utilities of 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]
- Harder… [1, 2, 3] or [3, 1, 1]
- Infinite sequences? [1, 2, 1, …] or [2, 1, 2, …]



Discounting

- It’s reasonable to maximize the sum of rewards
- It’s also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

1

Worth Now

γ

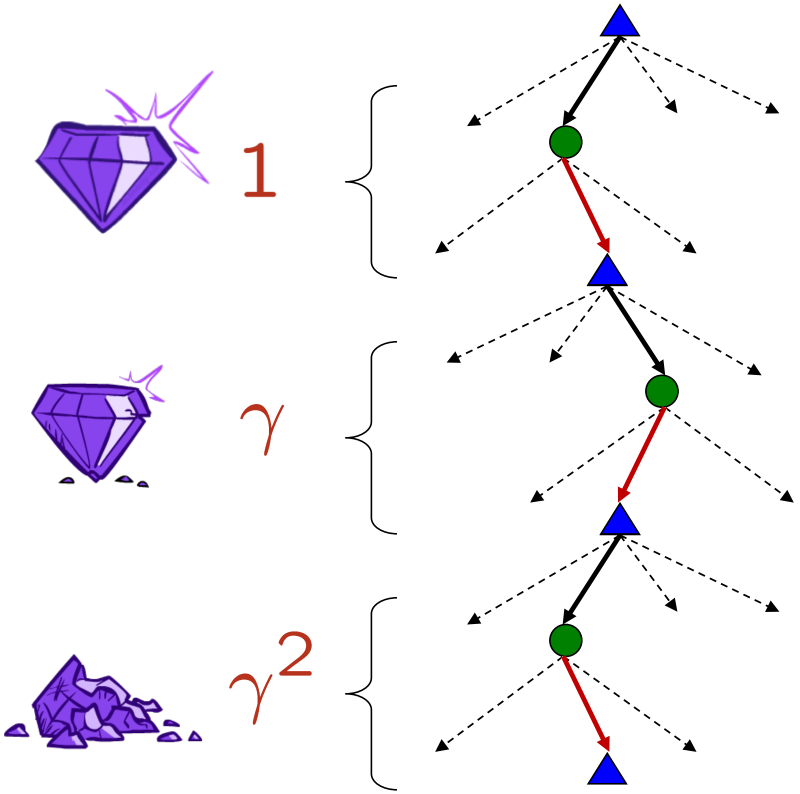
Worth Next Step

γ^2

Worth In Two Steps

Discounting

- How to discount?
  - Each time we descend a level, we multiply by the discount
- 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 = 2.75
  - U([3,1,1]) = 1*3 + 0.5*1 + 0.25*1 = 3.75
  - U([1,2,3]) < U([3,1,1])



Quiz: Discounting

- Given:

10				1
a	b	c	d	e

  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

- Quiz 1: For $\gamma = 1$, what is the optimal policy?

10	←	←	←	1

- Quiz 2: For $\gamma = 0.1$, what is the optimal policy?

10	←	←	→	1

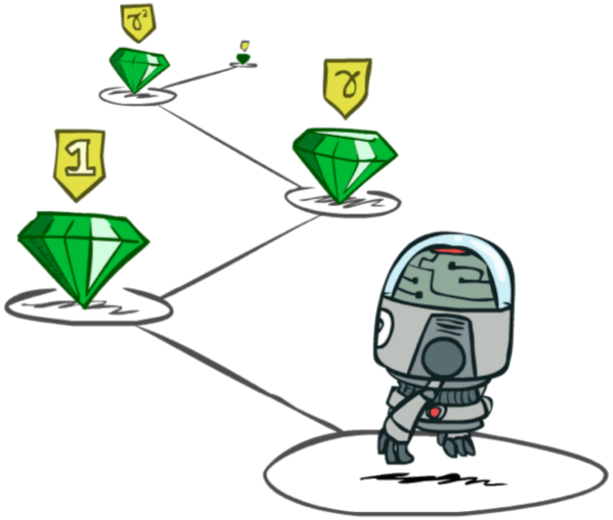
Stationary Preferences

- Theorem: if we assume stationary preferences:

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$

$$\Updownarrow$$

$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$



- Then: there are ***only two ways*** to define utilities
  - Additive utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$
  - Discounted utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots$

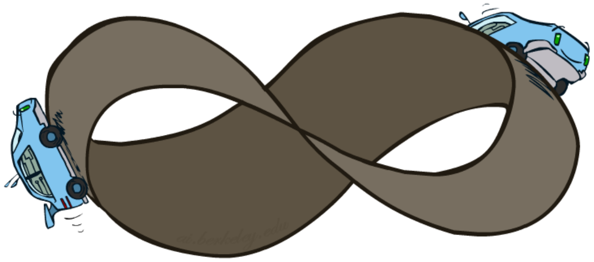
Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:

1. **Discounting:** use 0 < γ < 1

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

Smaller γ means smaller "horizon" – shorter term focus



2. **Finite horizon:** (similar to depth-limited search)

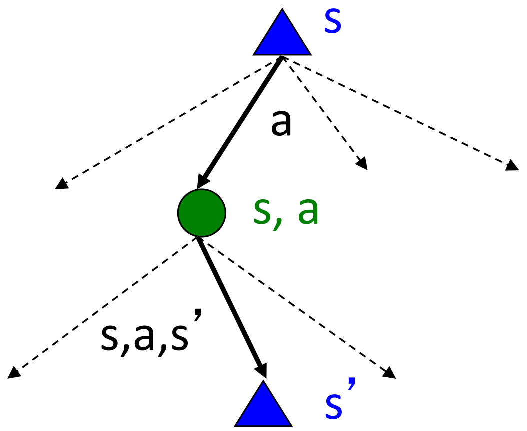
Add utilities, but terminate episodes after a fixed T-steps lifetime

Gives non-stationary policies (π depends on time left)

3. **Absorbing state:** guarantee that for every policy, a terminal state (like "overheated" for racing) will eventually be reached (eg. If ***every*** action had a chance of overheating)

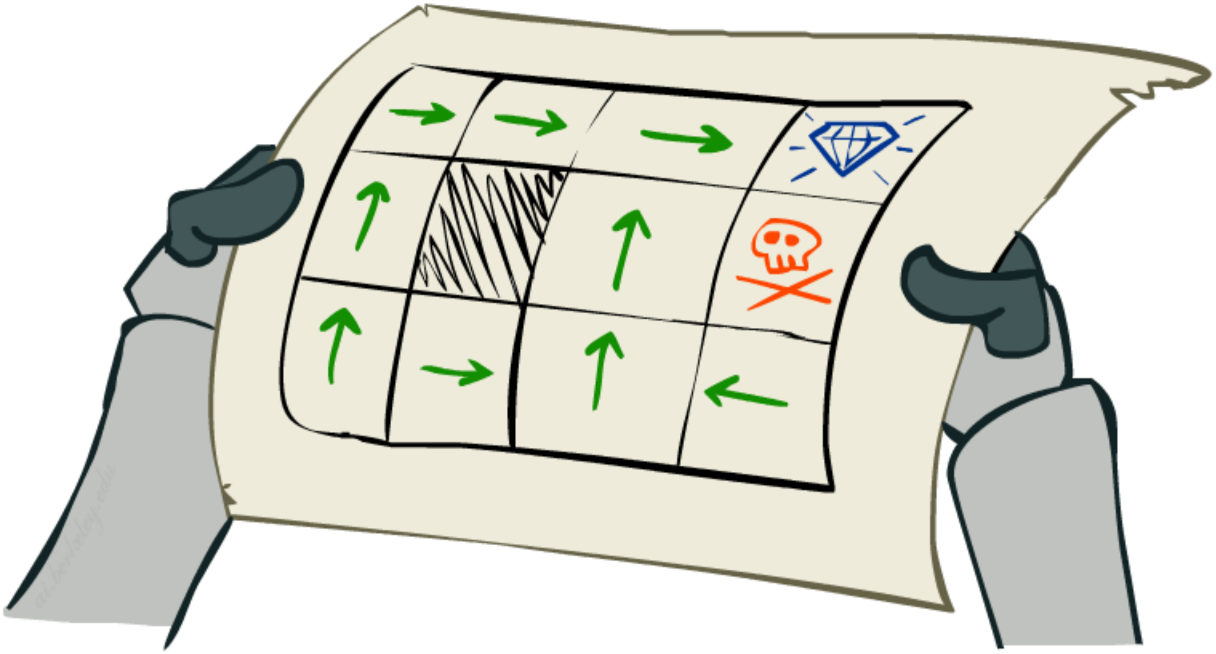
Recap: Defining MDPs

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

Solving MDPs



- Value Iteration
  - Asynchronous VI
  - RTDP
  - Etc...
- Policy Iteration
- Reinforcement Learning

π* Specifies The Optimal Policy

$\pi^*(s)$ = optimal action from state s

V* = Optimal Value Function

The value (utility) of a state s:

$V^*(s)$

“expected utility starting in s & acting optimally forever”

Equivalently: “value of s, following π^* forever”

Q*

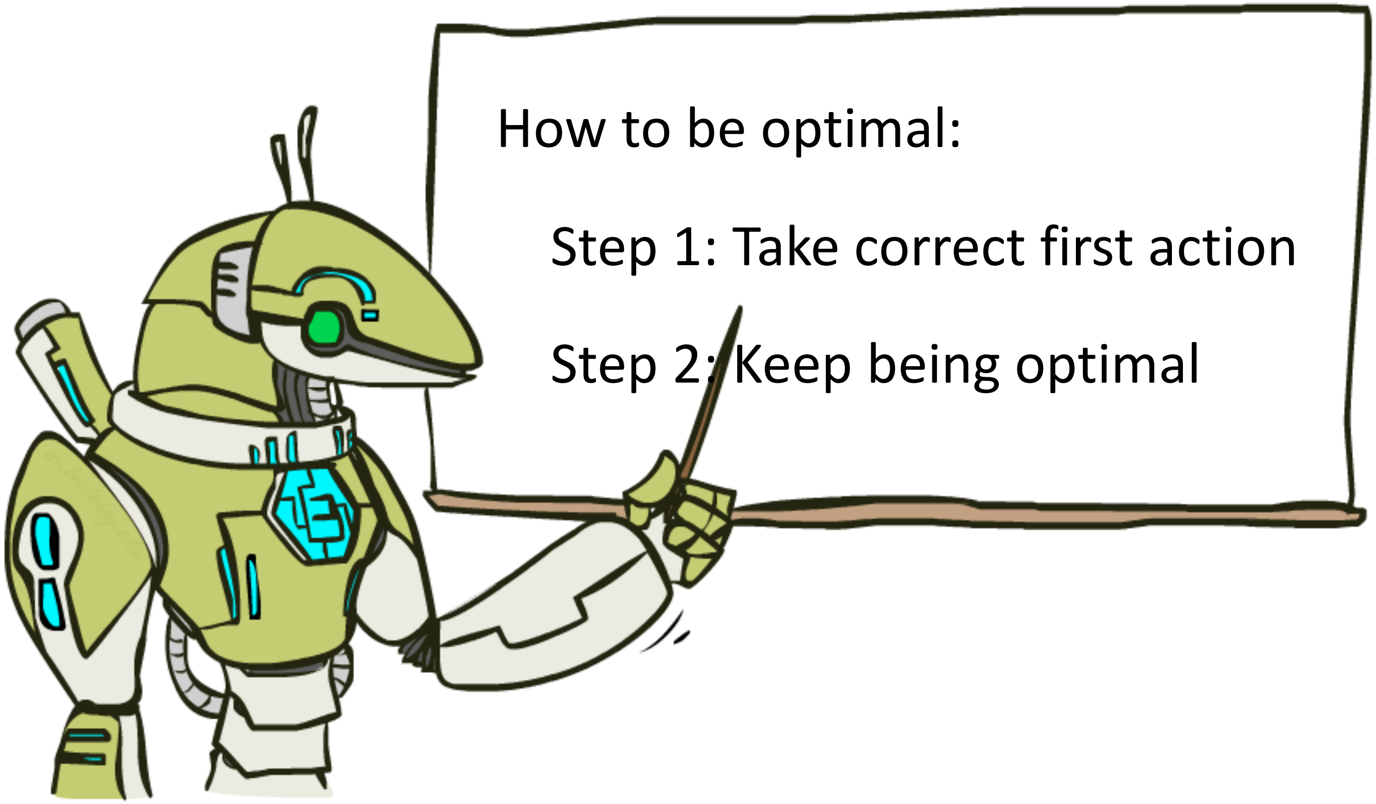
The value (utility) of the q-state (s,a):

$Q^{*}(s,a)$

“expected utility of 1) starting in state ***s***
2) first taking action ***a***
3) acting *optimally* (ala π^{*}) forever after that”

Q*(s,a) = reward from executing a in s then ending in s’
plus… discounted value of V*(s’)

The Bellman Equations

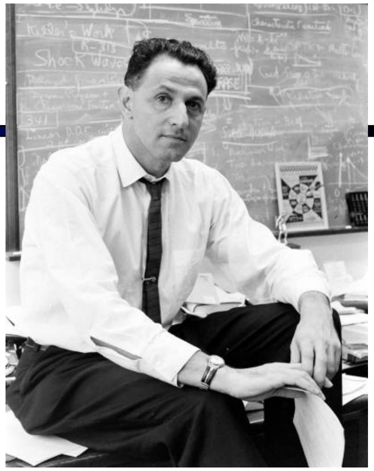


How to be optimal:

Step 1: Take correct first action

Step 2: Keep being optimal

The Bellman Equations

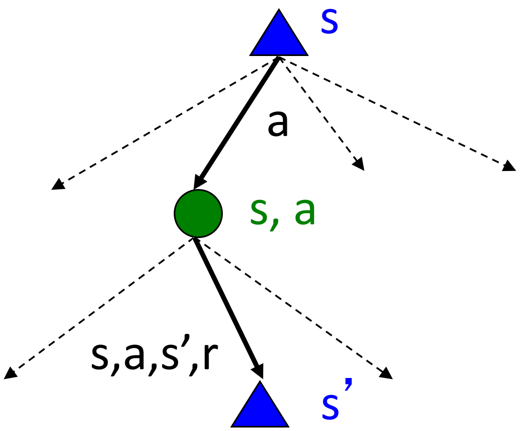


(1920-1984)

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

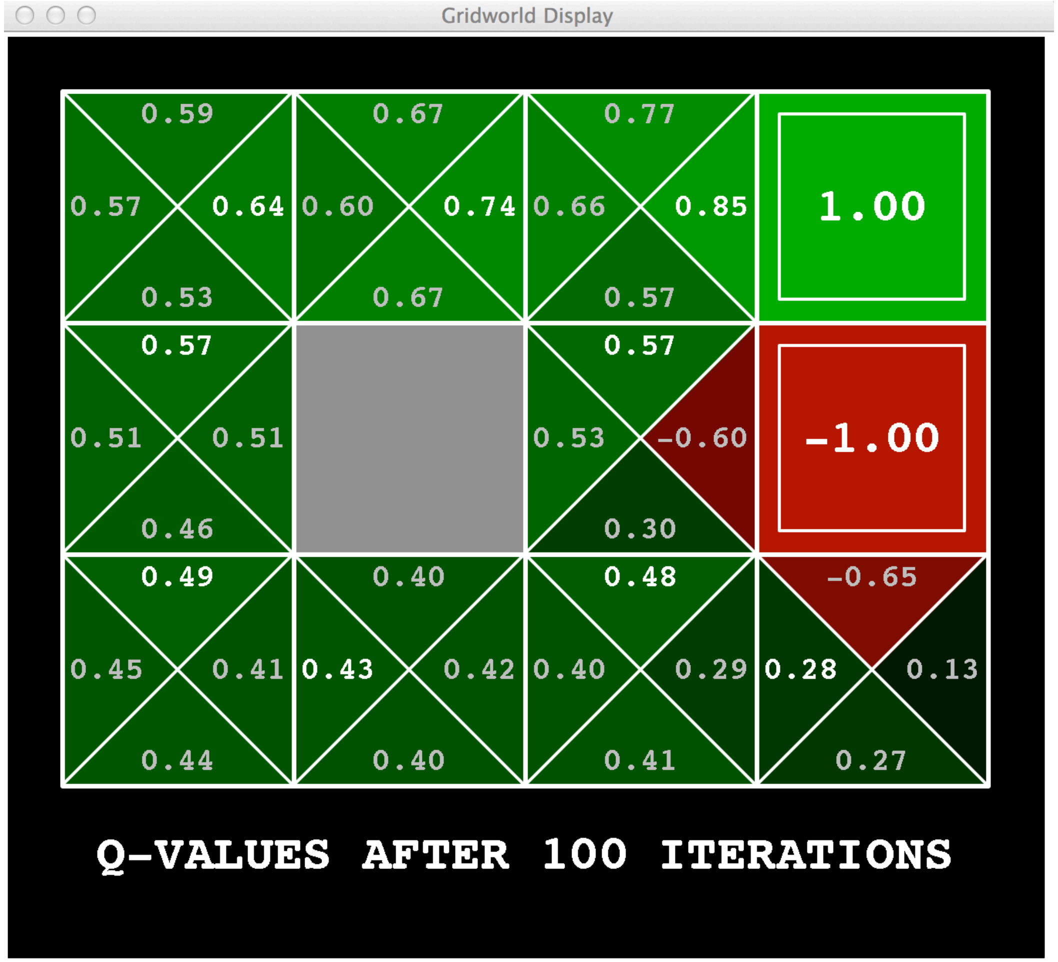
$$V^*(s) = \max_a Q^*(s,a)$$

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



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

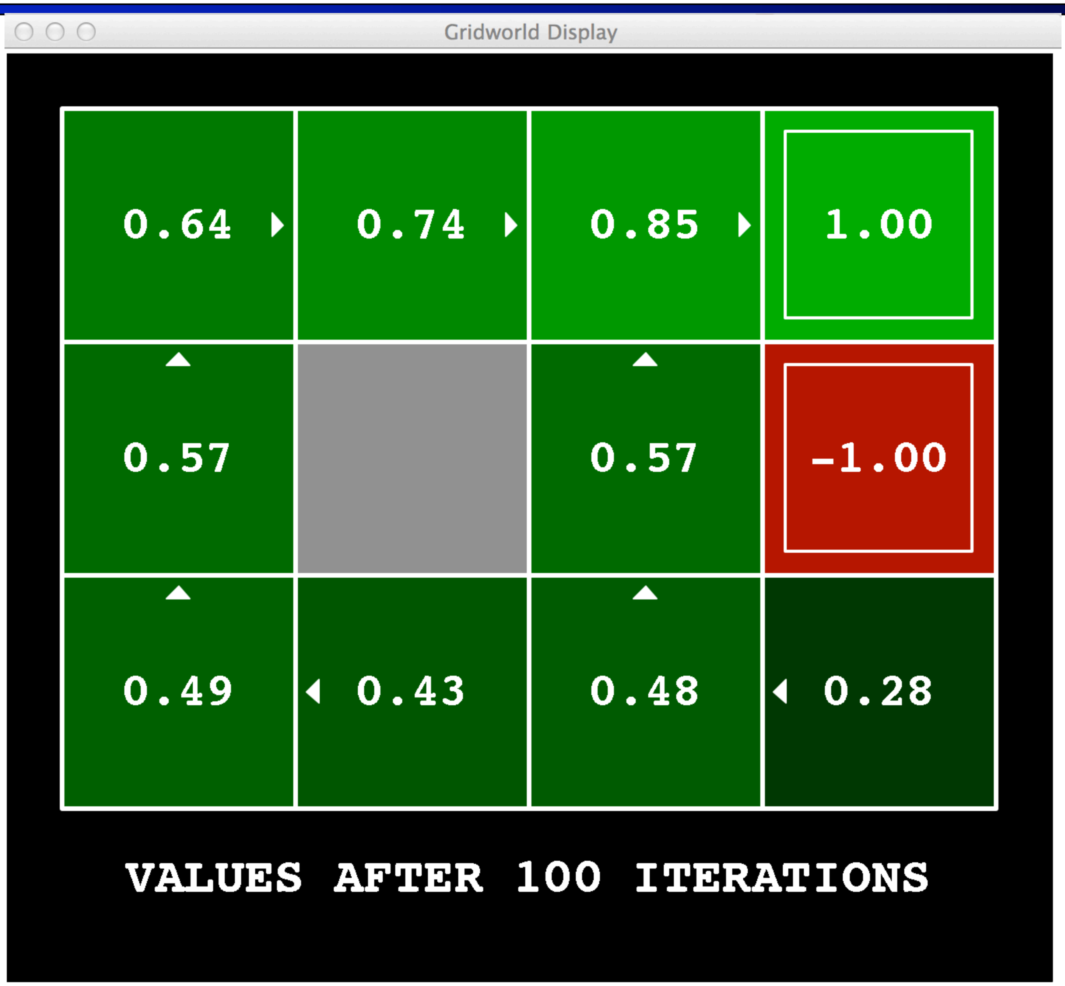
Gridworld: Q*



Q-VALUES AFTER 100 ITERATIONS

Gridworld Values V*

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

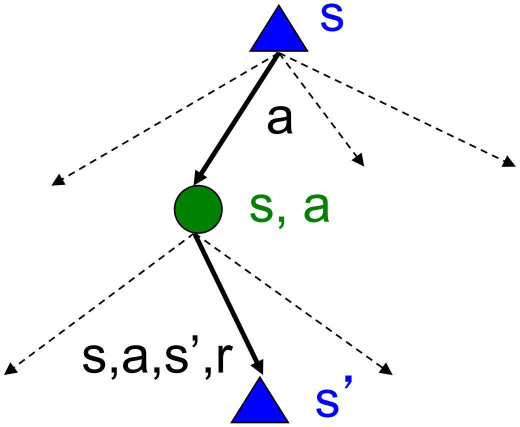


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.57		▴ 0.57	-1.00
▴ 0.49	◂ 0.43	▴ 0.48	◂ 0.28

VALUES AFTER 100 ITERATIONS

Values of States

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!



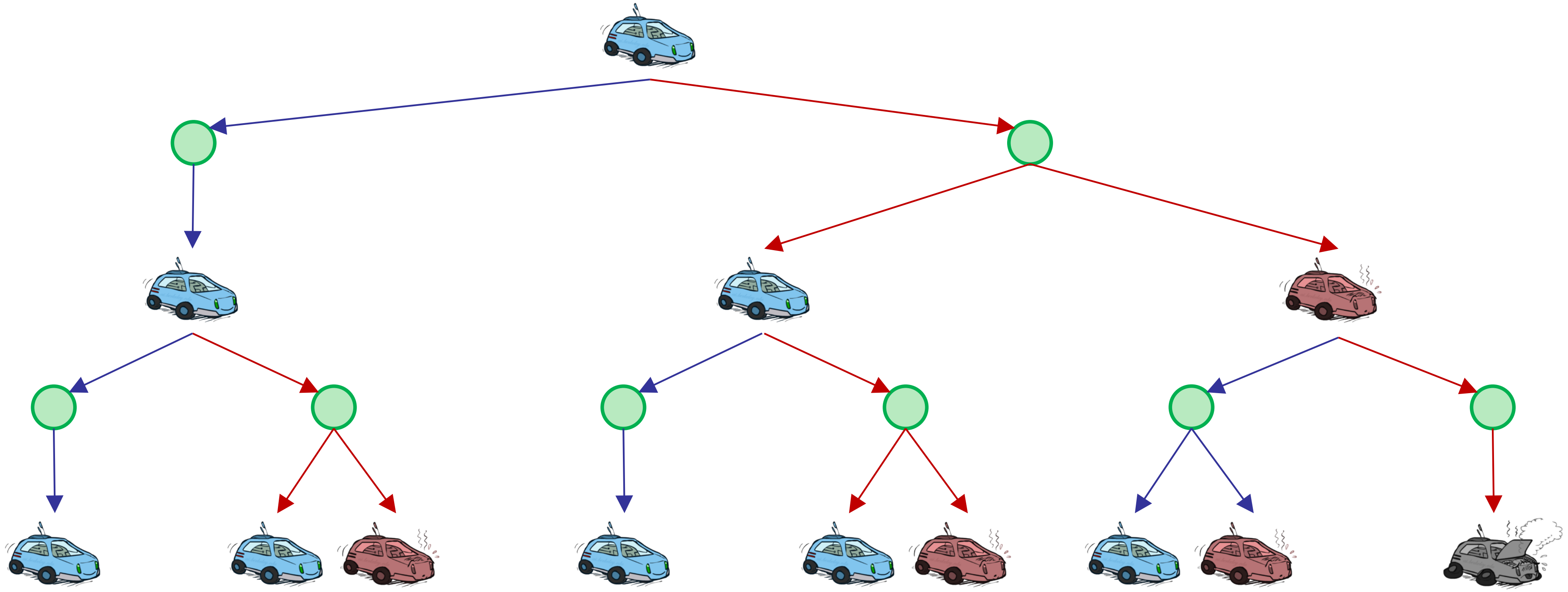
- Recursive definition of value:

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

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

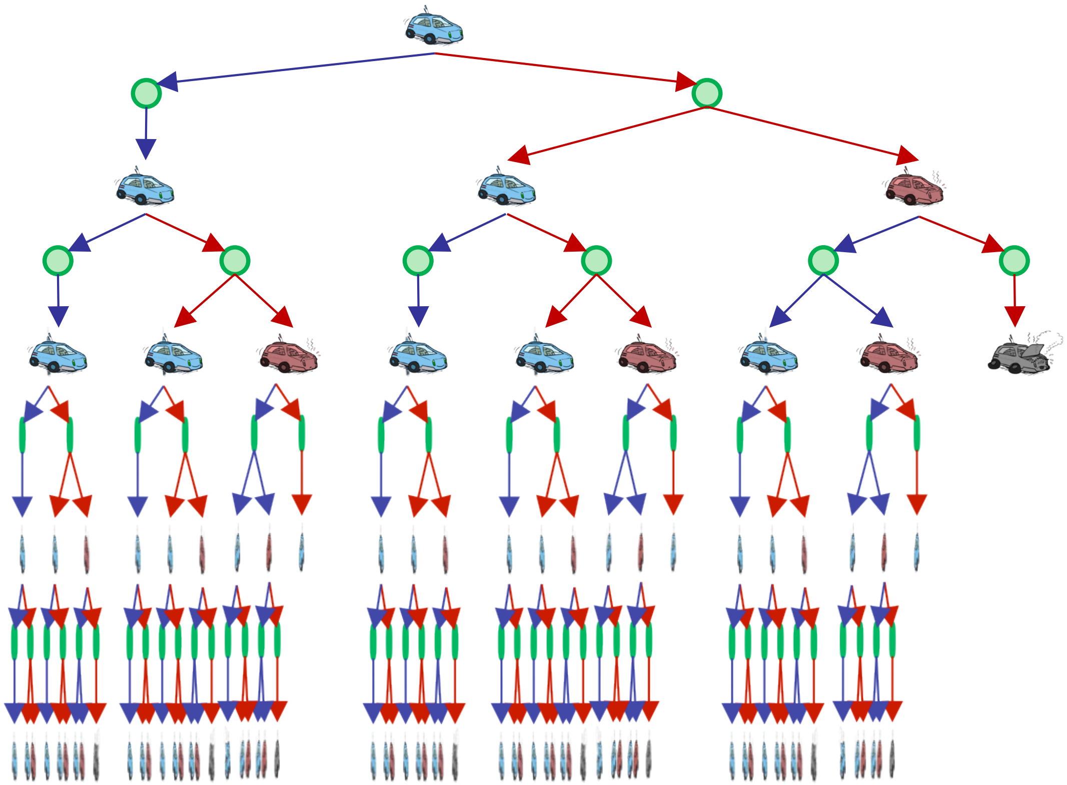
i.e. $$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

Racing Search Tree



No End in Sight…

- Problem 1: Tree goes on forever
  - Rewards @ each step → **V changes**
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree *eventually* don't matter much (< ε) if $\gamma < 1$

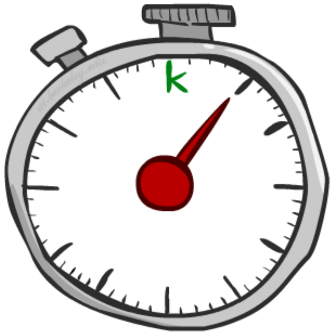


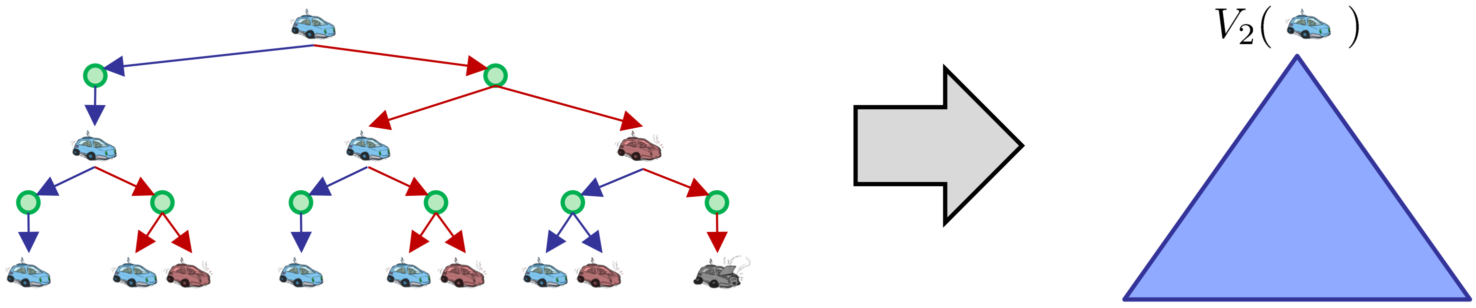
- Problem 2: Too much repeated work
  - Idea: Only compute needed quantities once
  - Like **graph search** (*vs.* tree search)
  - Ako dynamic programming

Time-Limited Values

- Key idea: ***time-limited values***

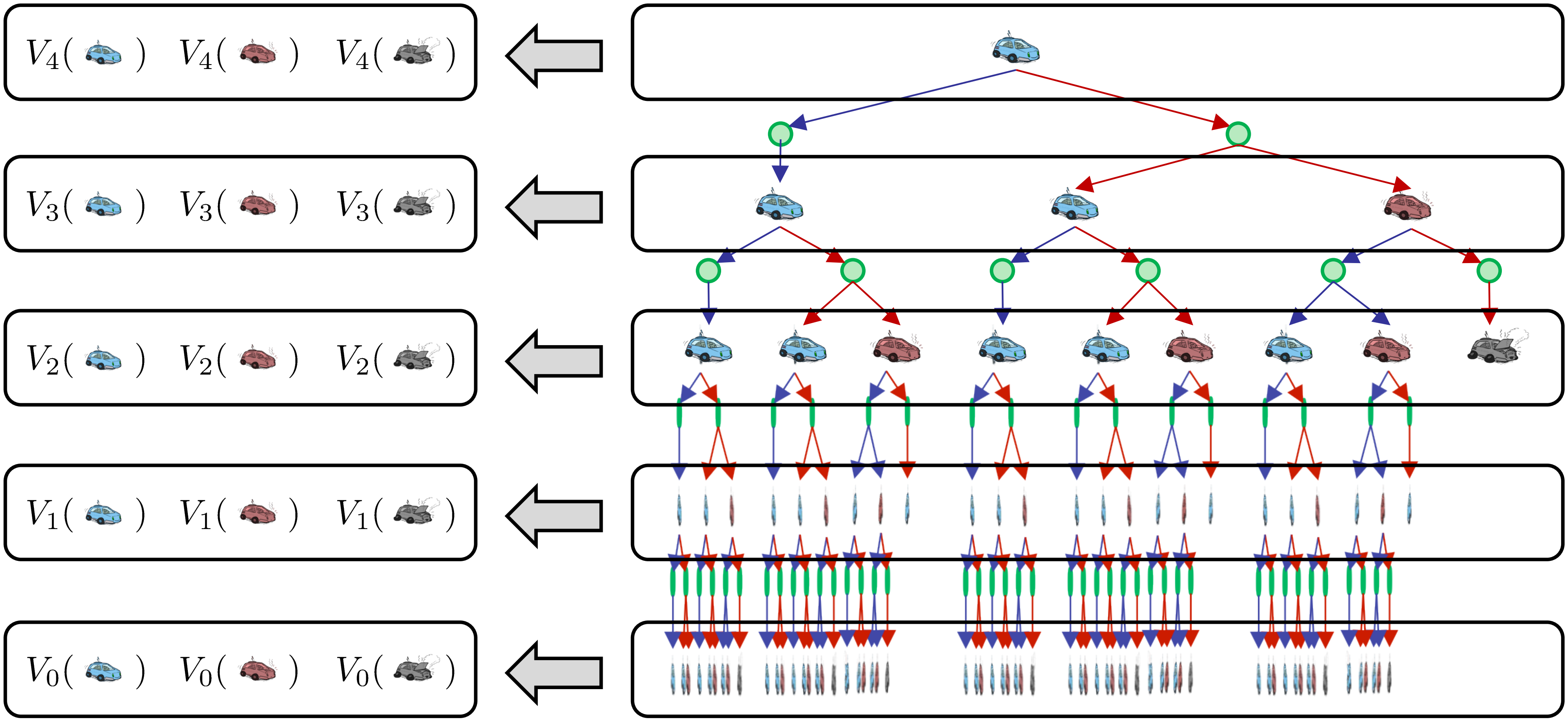
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
  - Equivalently, it's what a depth-k expectimax would give from s





[Demo – time-limited values (L8D6)]

Time-Limited Values: Avoiding Redundant Computation



$V_4(\ )$ $V_4(\ )$ $V_4(\ )$

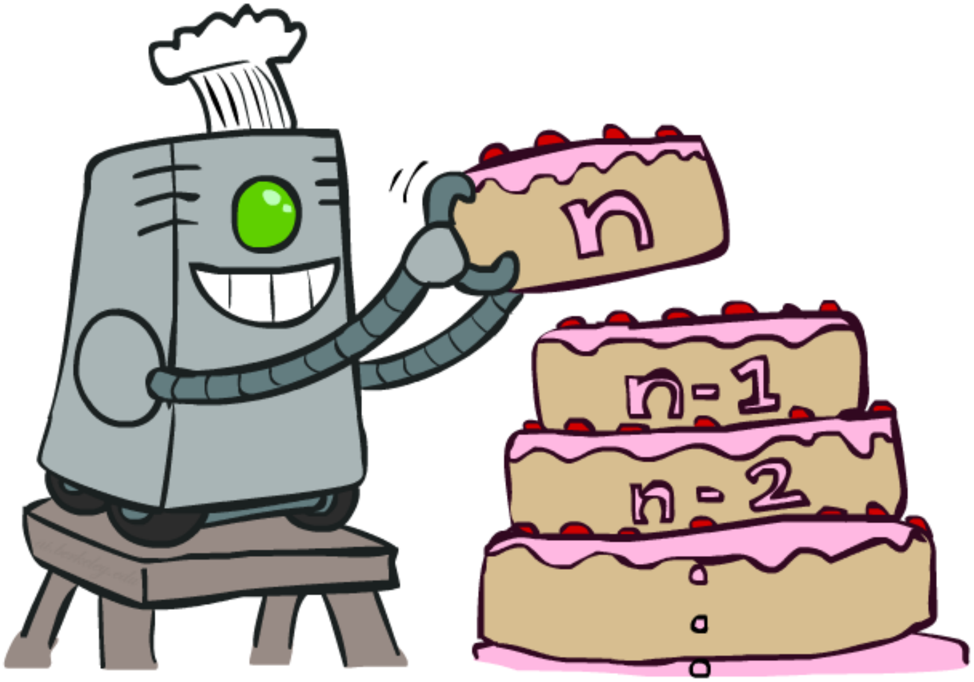
$V_3(\ )$ $V_3(\ )$ $V_3(\ )$

$V_2(\ )$ $V_2(\ )$ $V_2(\ )$

$V_1(\ )$ $V_1(\ )$ $V_1(\ )$

$V_0(\ )$ $V_0(\ )$ $V_0(\ )$

Value Iteration



Value Iteration

Called a
“Bellman Backup”

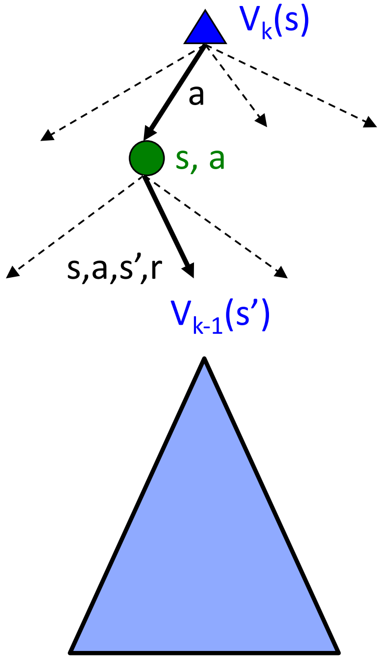
- **Forall s, tialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat**

  K += 1

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

  $V_k(s) = \text{Max}_a Q_k(s, a)$

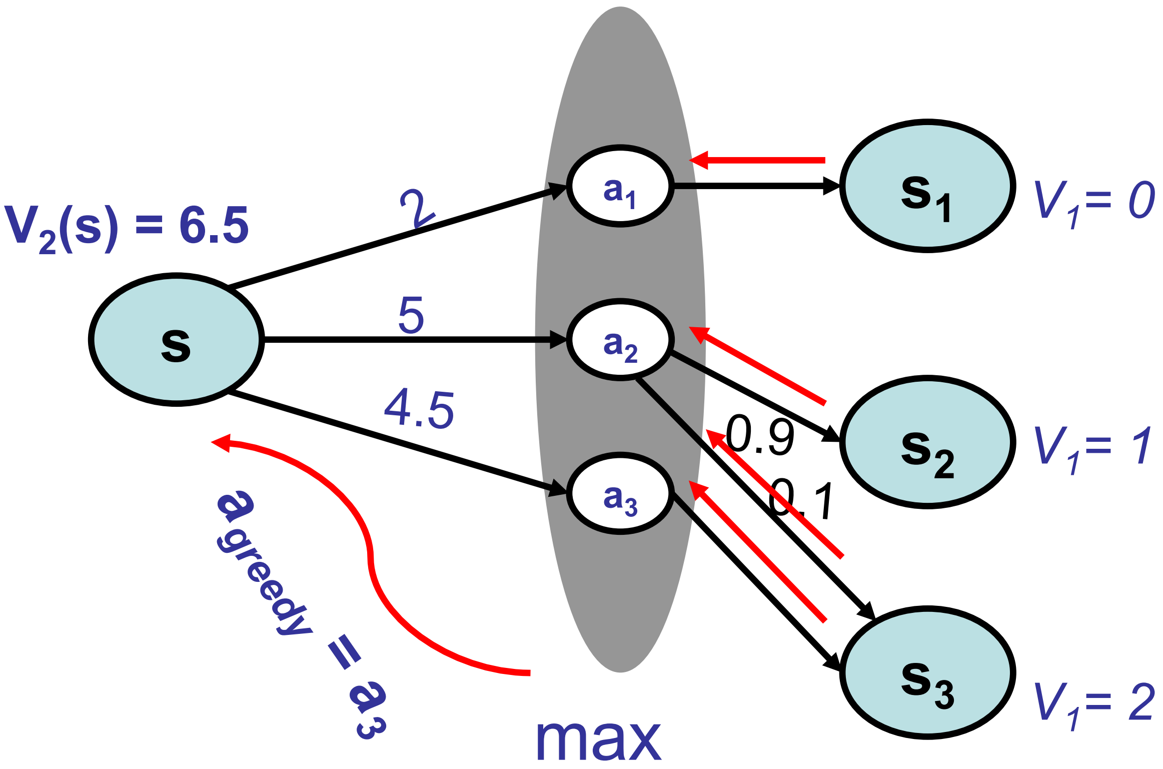
  } do $\forall$ s, a
- **Repeat until $|V_k(s) - V_{k-1}(s)| < \varepsilon$, forall s** *“convergence”*



Successive approximation; dynamic programming

Example: Bellman Backup

Assume γ ~ 1



$Q_1(s,a_1) = 2 + \gamma\ 0$

~ 2

$Q_1(s,a_2) = 5 + \gamma\ 0.9 \sim 1$

$+ \gamma\ 0.1 \sim 2$

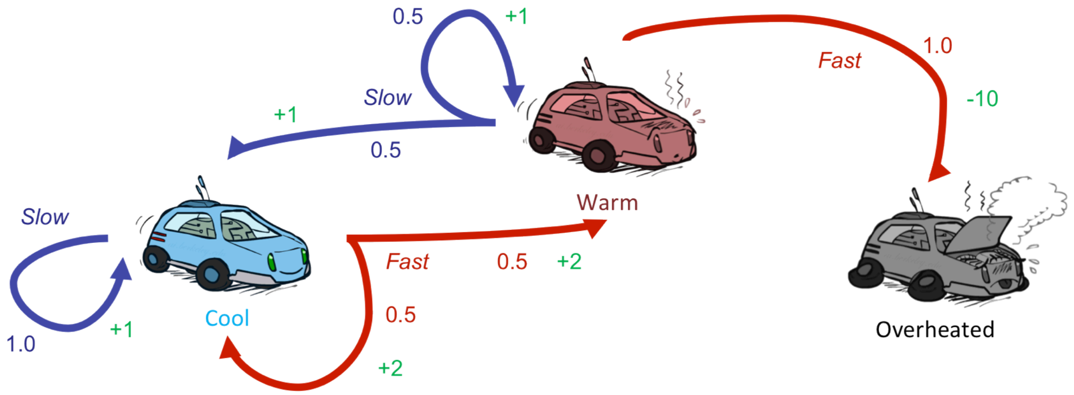
~ 6.1

$Q_1(s,a_3) = 4.5 + \gamma\ 2$

~ 6.5

Example: Value Iteration

Assume no discount (gamma=1) to keep math simple!



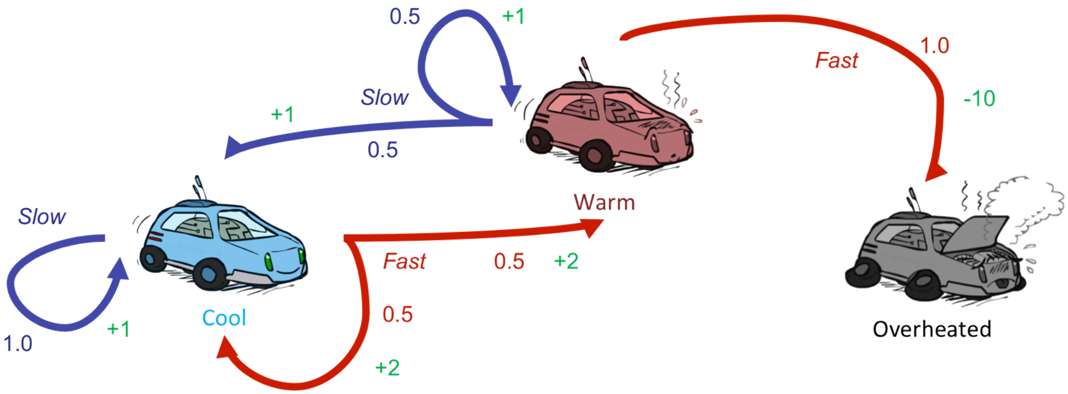
$$Q_k(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_{k-1}(s')]$$

$$V_k(s) = \text{Max}_{a} Q_k (s, a)$$

Example: Value Iteration

Assume no discount (gamma=1) to keep math simple!

V_0	0	0	0
V_1			
V_2			



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

$$V_k(s) = \text{Max}_{a} Q_k (s, a)$$

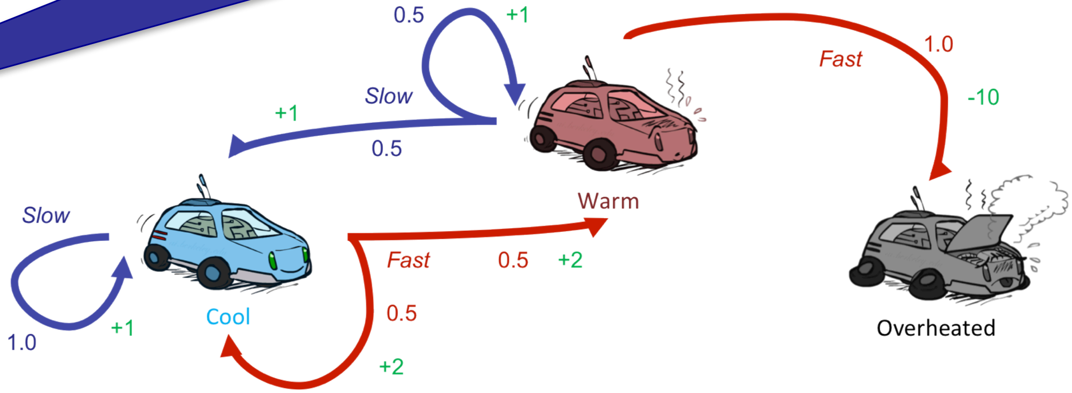
Example: Value Iteration

Q(,fast) =

Q(slow) =

o math simple!

V_0	0	0	0
$Q_1(s,a)$=			0
V_1			0
V_2			



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

$$V_k(s) = \text{Max}_a Q_k (s, a)$$

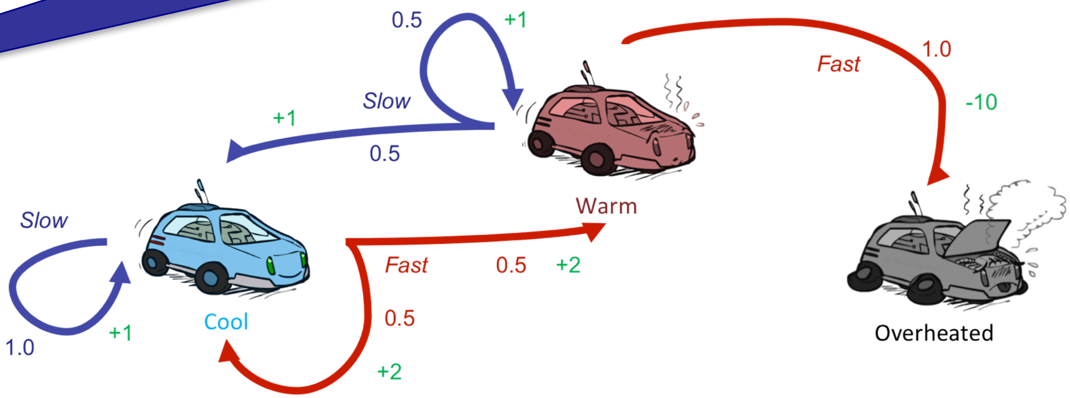
Example: Value Iteration

math simple!

Q(,fast) = -10 + 0

Q(slow) = ½(1 + 0) + ½(1+0)

V_0	0	0	0
$Q_1(s,a)=$		1, -10	0
V_1		1	0
V_2			



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

$$V_k(s) = \text{Max}_a Q_k(s, a)$$

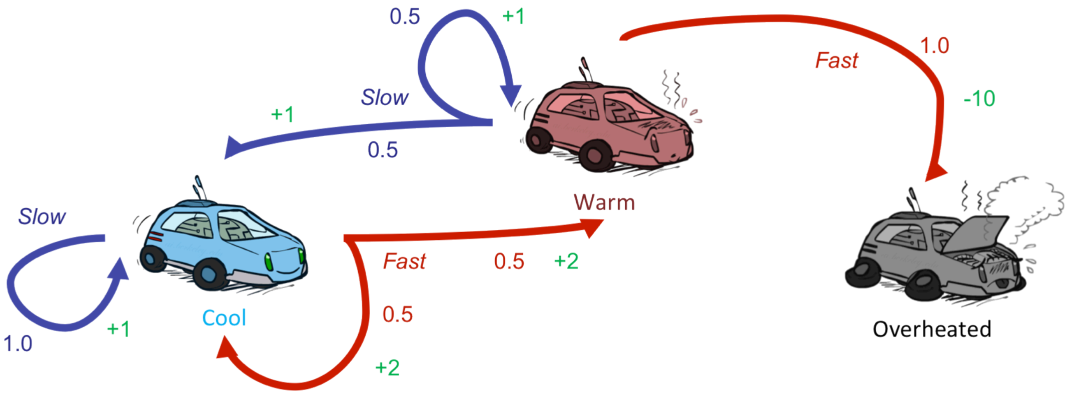
Exa n

ma=1) to keep math simple!

Q(, fast) = ½(2 + 0) + ½(2 + 0)

Q(, slow) = 1*(1 + 0)

V_0	0	0	0
$Q_1(s,a)$=	1, 2	1,-10	0
V_1	2	1	0
V_2			



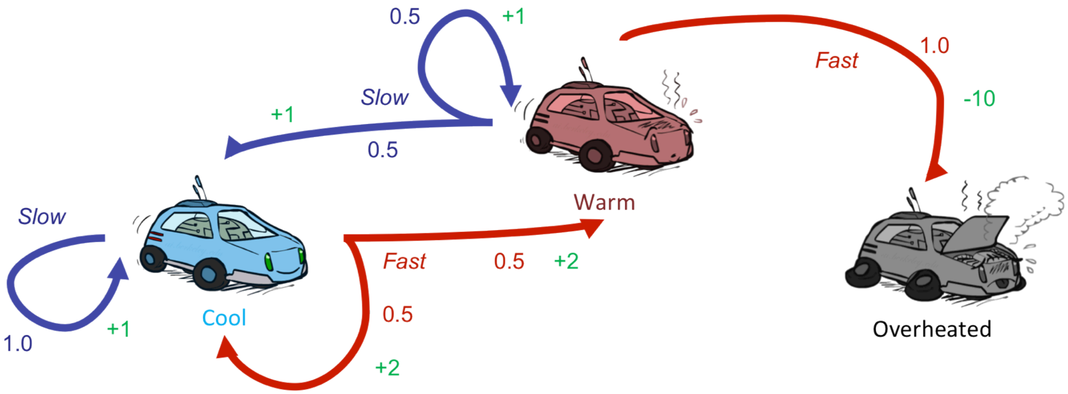
$$Q_k(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_{k-1}(s')]$$

$$V_k(s) = \text{Max}_a\, Q_k (s, a)$$

Example: Value Iteration

Assume no discount (gamma=1) to keep math simple!

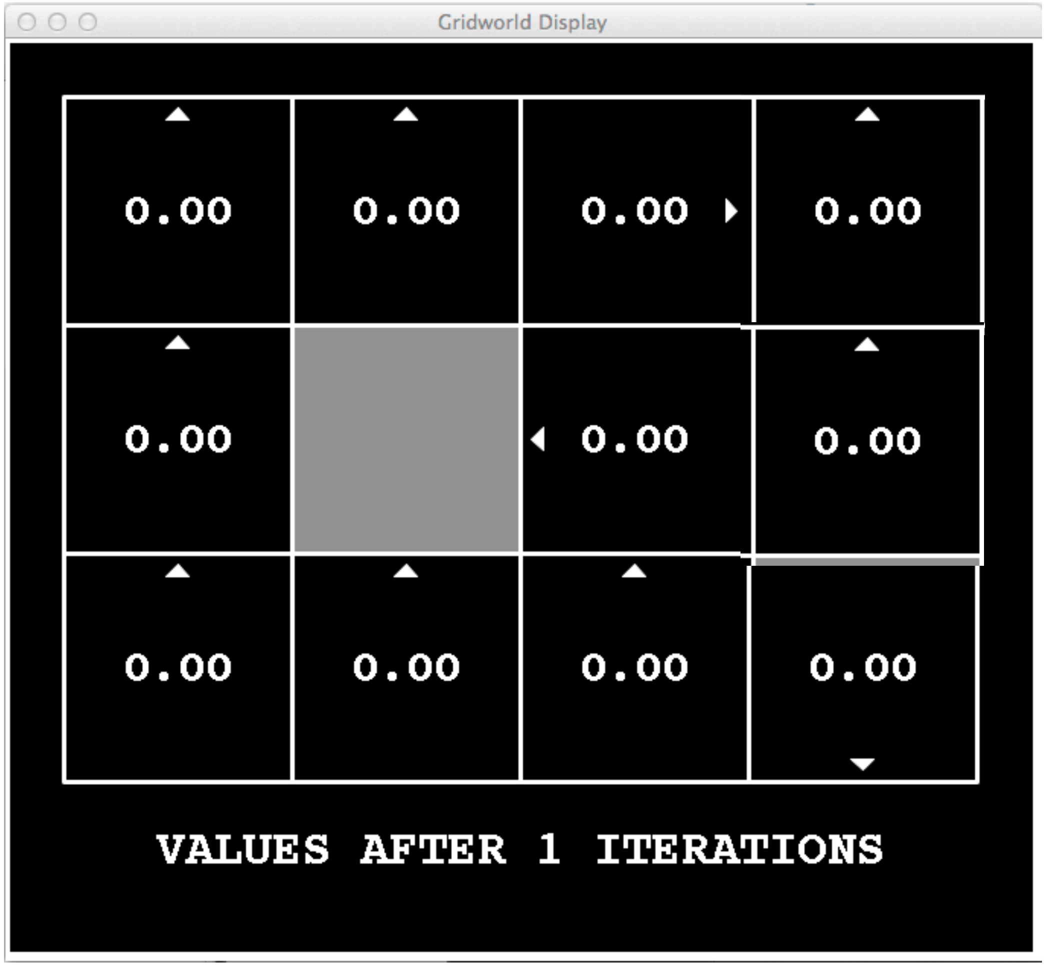
V_0	0	0	0
$Q_1(s,a)=$	1, 2	1,-10	0
V_1	2	1	0
$Q_2(s,a)=$	3,3.5	2.5,-10	0
V_2	3.5	2.5	0



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

$$V_k(s) = \text{Max}_{a} Q_k (s, a)$$

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 1 ITERATIONS

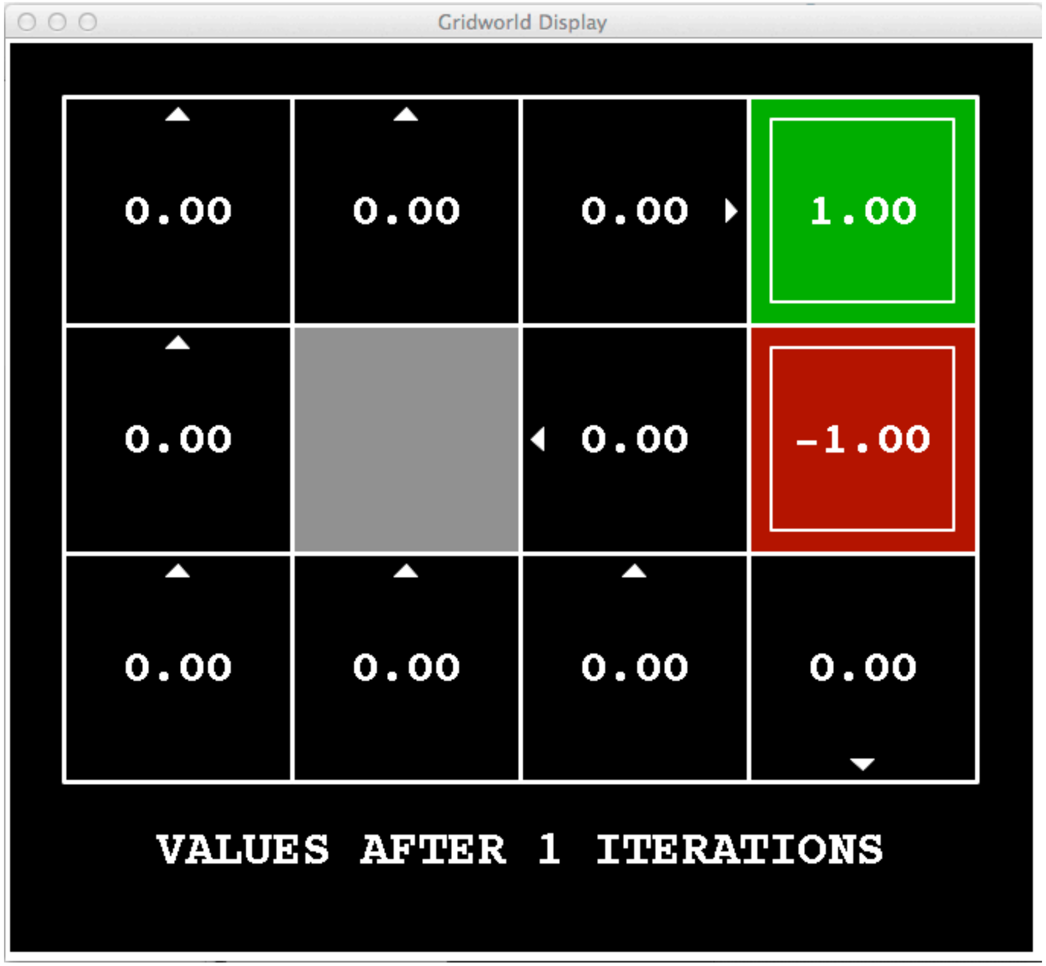
Noise = 0.2
Discount = 0.9
Living reward = 0

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.

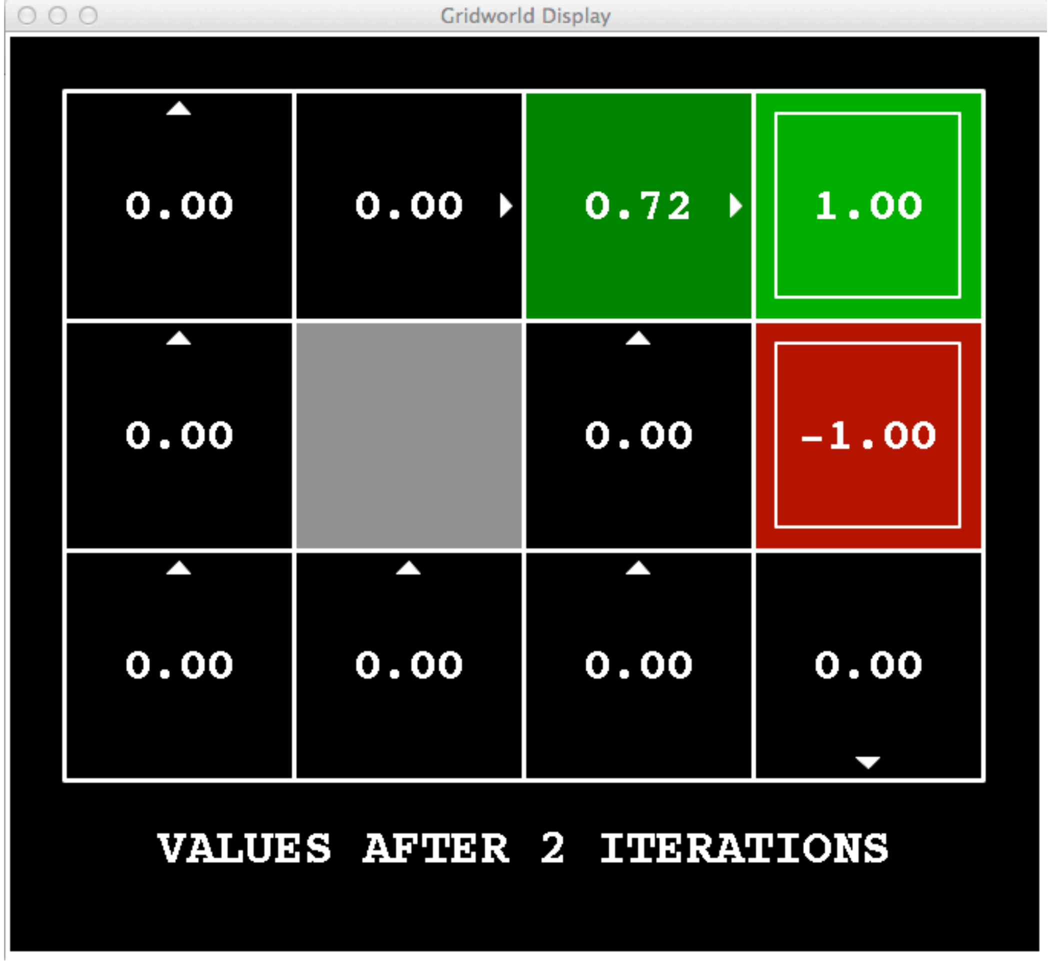


▲ 0.00	▲ 0.00	0.00 ▶	1.00
▲ 0.00	▨	◀ 0.00	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 1 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

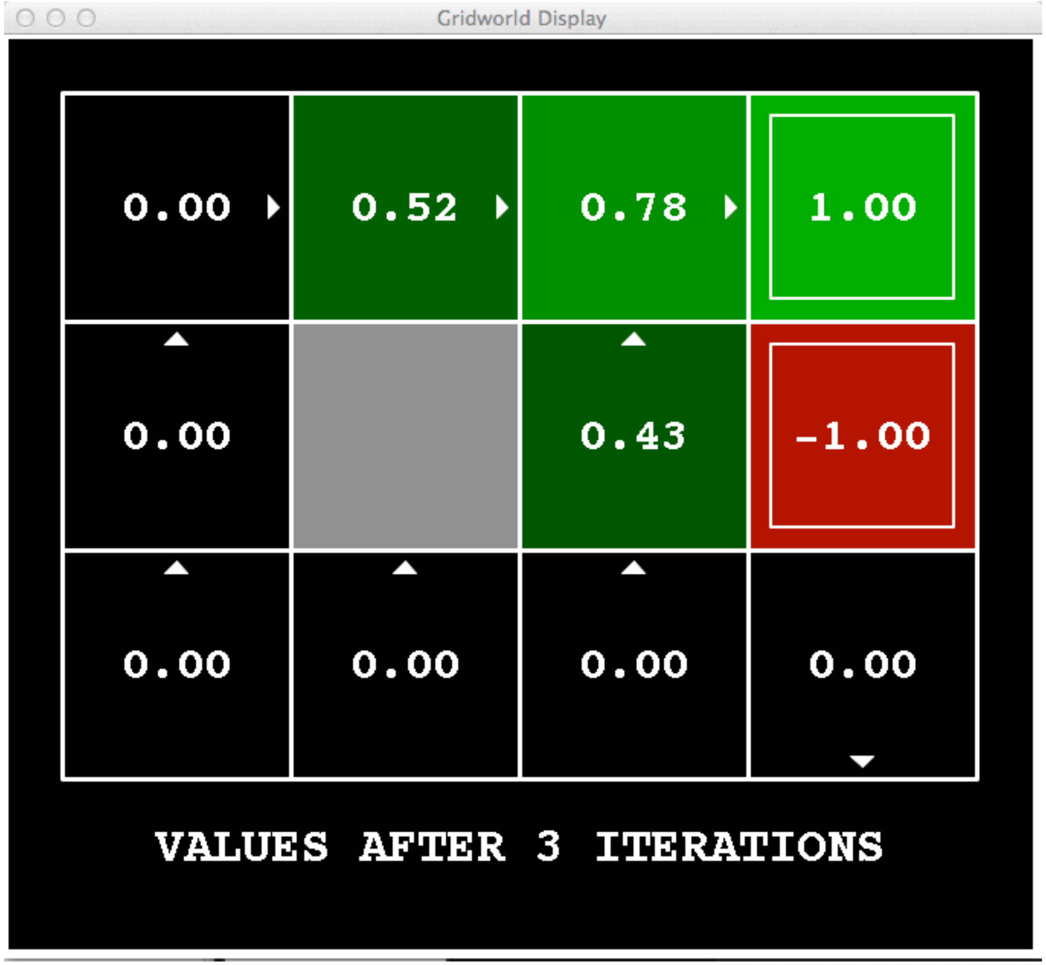


▲ 0.00	0.00 ▶	0.72 ▶	1.00
▲ 0.00		▲ 0.00	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 2 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

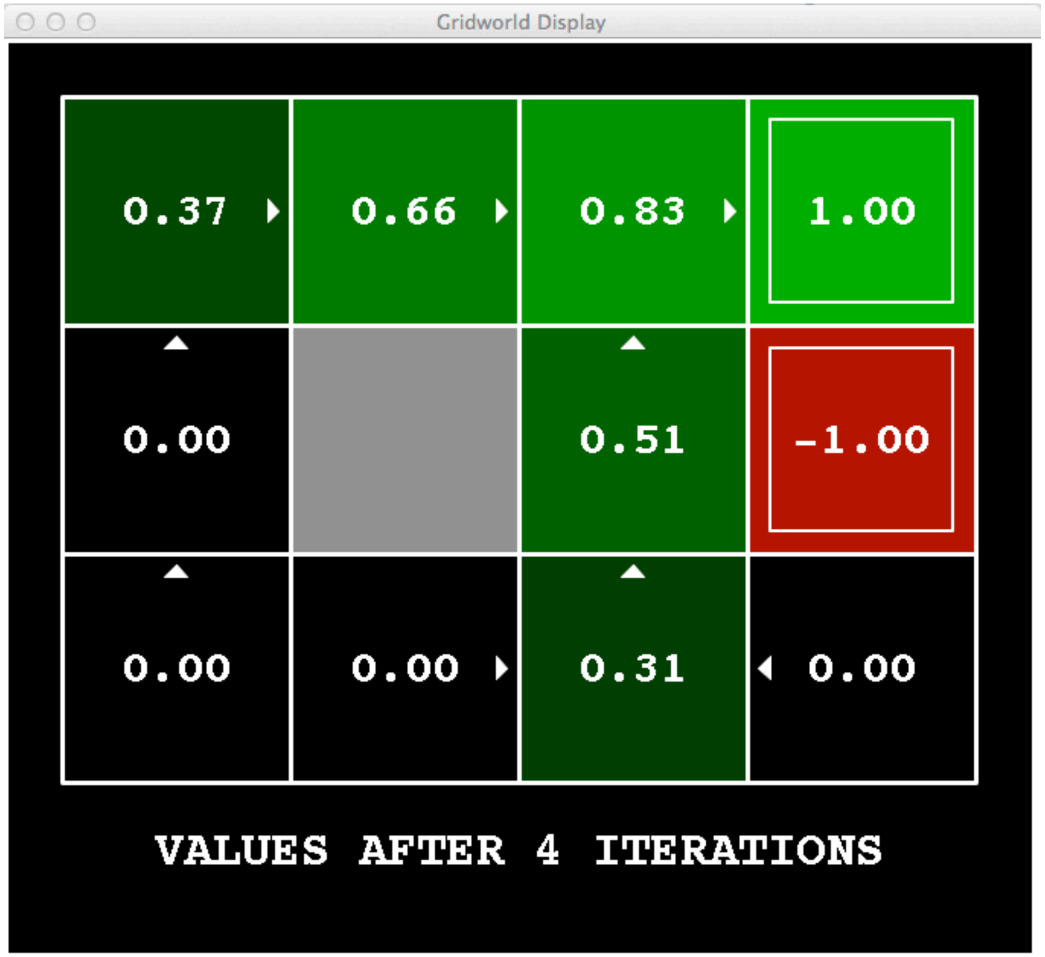


0.00 ▶	0.52 ▶	0.78 ▶	1.00
▲ 0.00		▲ 0.43	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 3 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

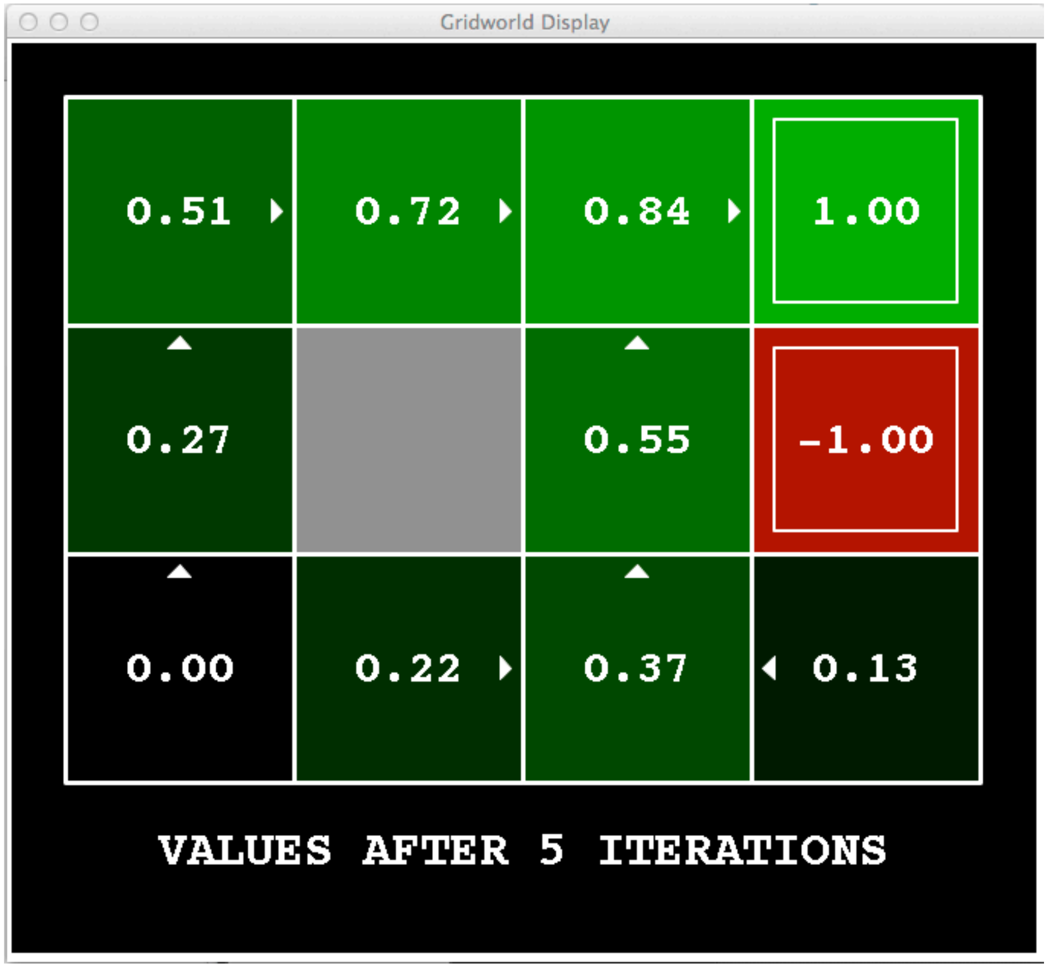


0.37 ▶	0.66 ▶	0.83 ▶	1.00
▲ 0.00		▲ 0.51	-1.00
▲ 0.00	0.00 ▶	▲ 0.31	◀ 0.00

VALUES AFTER 4 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=5

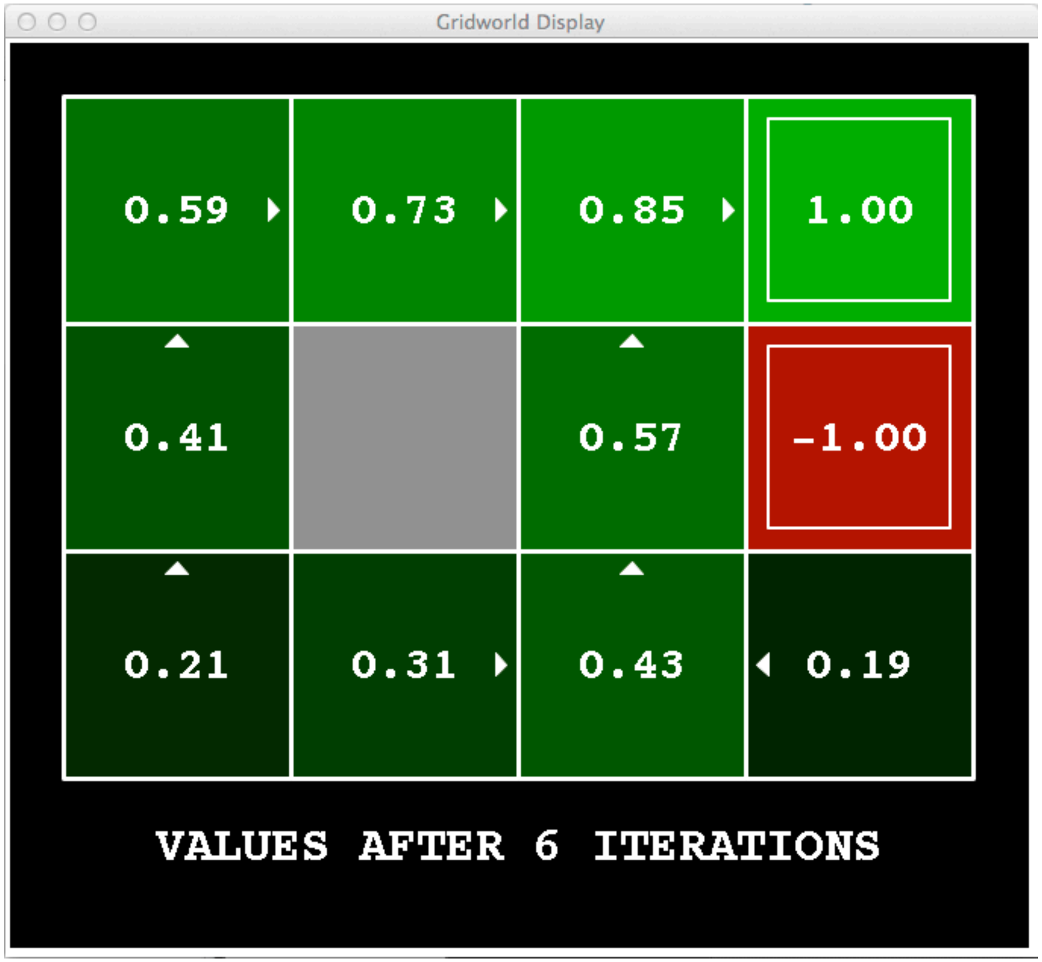


0.51 ▸	0.72 ▸	0.84 ▸	1.00
▴ 0.27		▴ 0.55	-1.00
▴ 0.00	0.22 ▸	▴ 0.37	◂ 0.13

VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=6

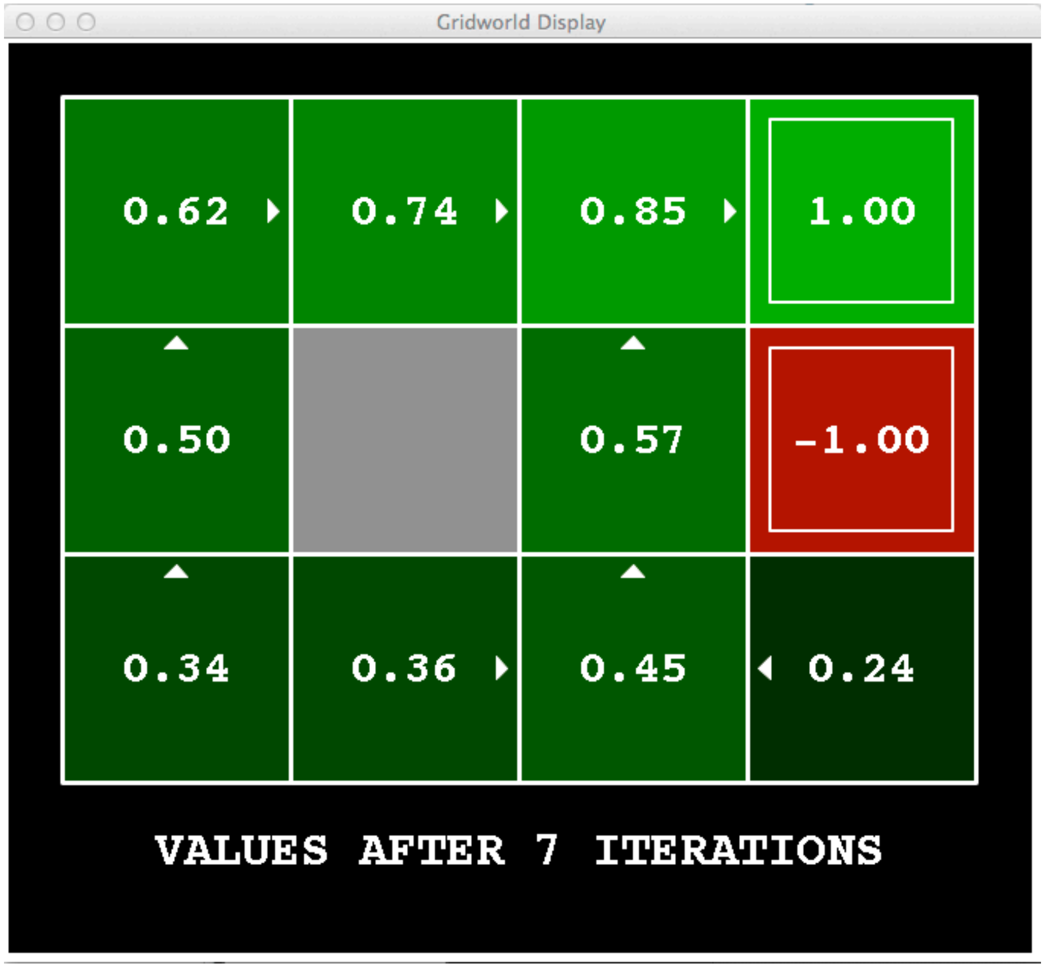


0.59 ▶	0.73 ▶	0.85 ▶	1.00
▲ 0.41		▲ 0.57	-1.00
▲ 0.21	0.31 ▶	▲ 0.43	◀ 0.19

VALUES AFTER 6 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=7

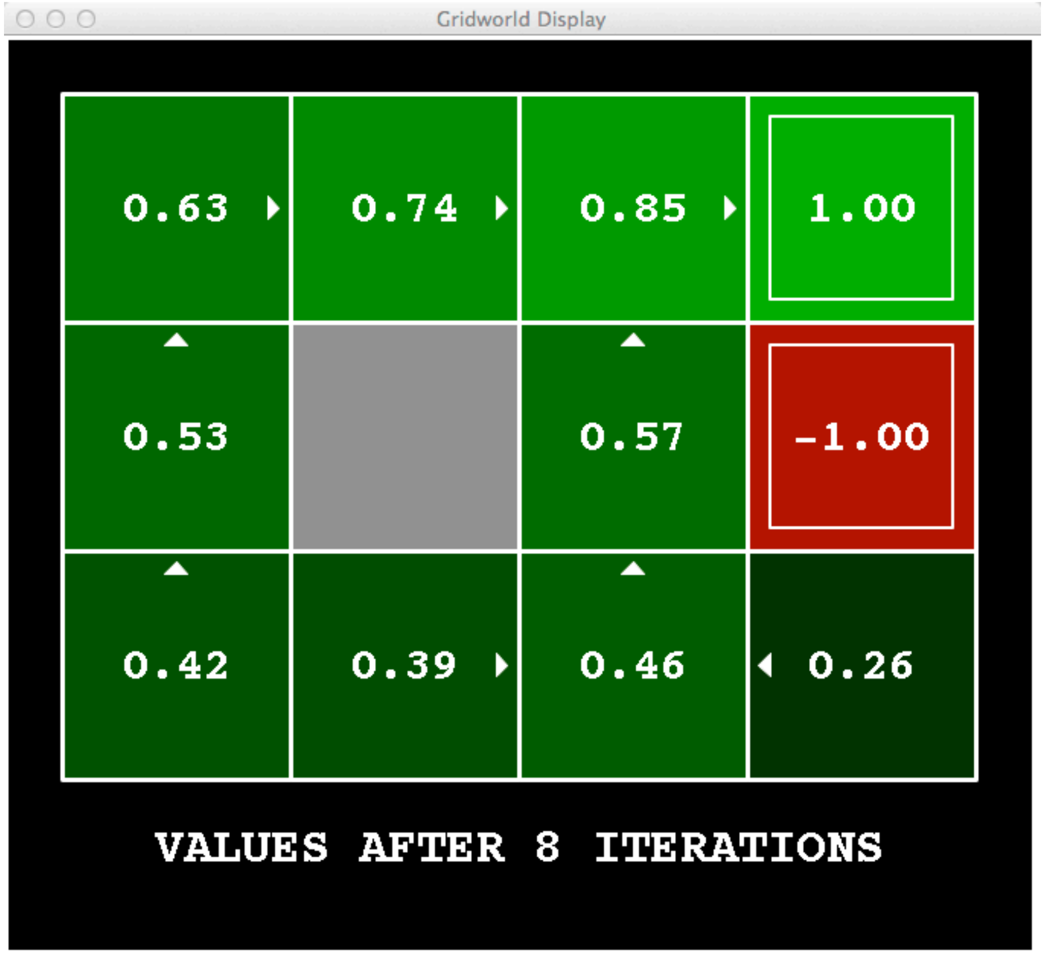


0.62 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.50		▴ 0.57	-1.00
▴ 0.34	0.36 ▸	▴ 0.45	◂ 0.24

VALUES AFTER 7 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=8

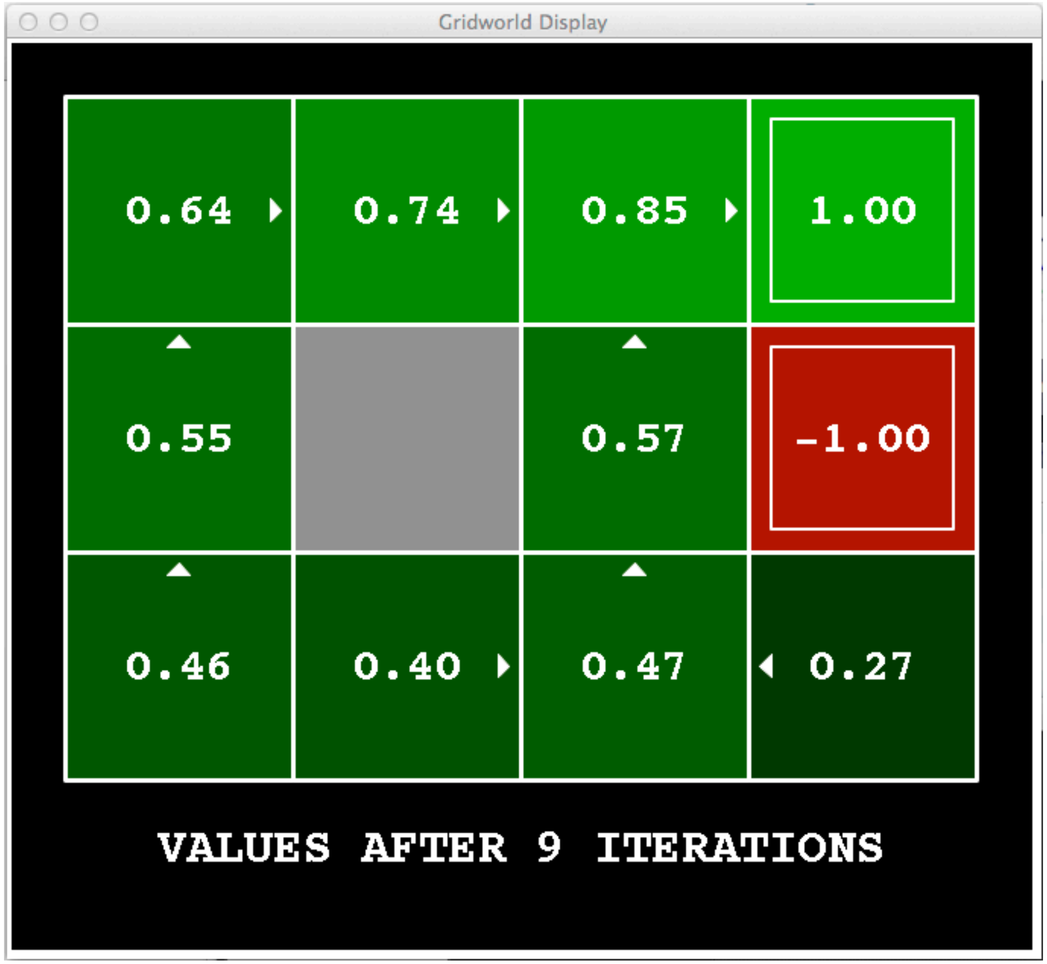


0.63 ▸	0.74 ▸	0.85 ▸	1.00
▲ 0.53		▲ 0.57	-1.00
▲ 0.42	0.39 ▸	▲ 0.46	◂ 0.26

VALUES AFTER 8 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=9

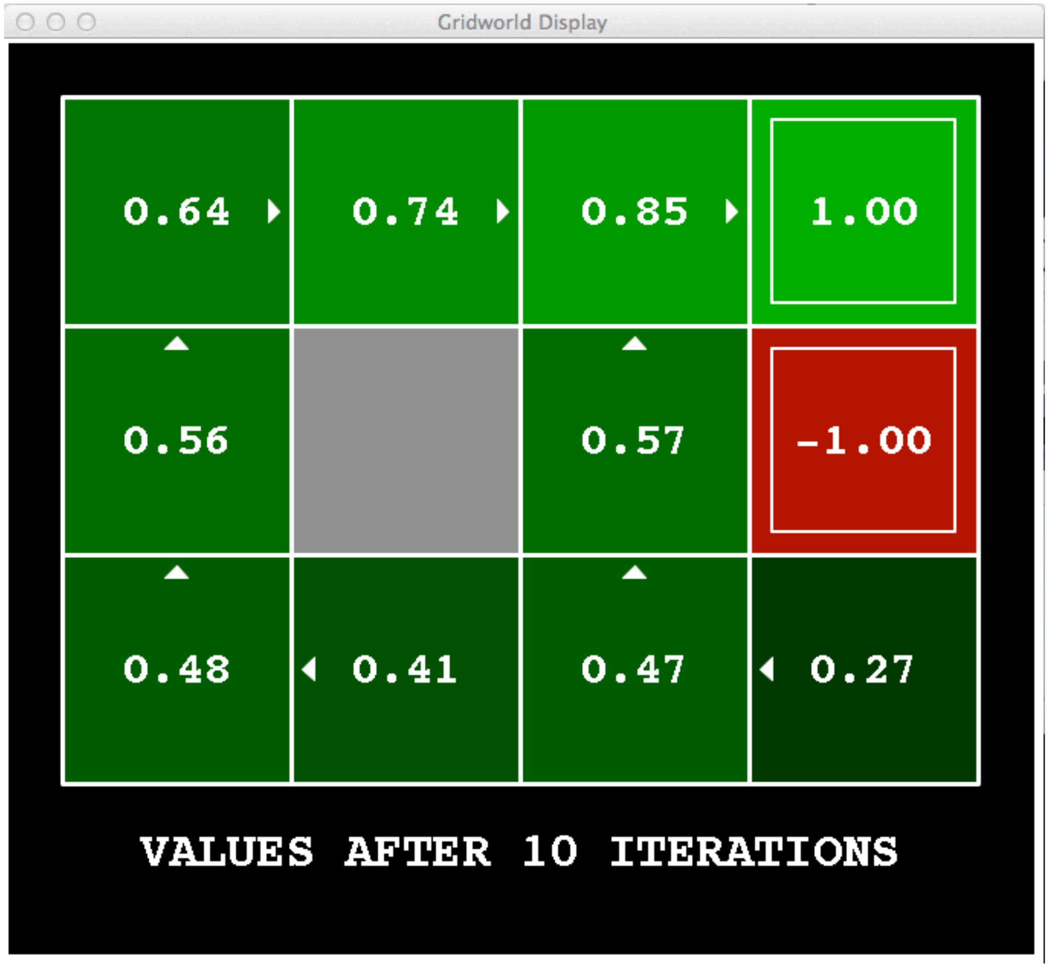


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.55		▴ 0.57	-1.00
▴ 0.46	0.40 ▸	▴ 0.47	◂ 0.27

VALUES AFTER 9 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=10

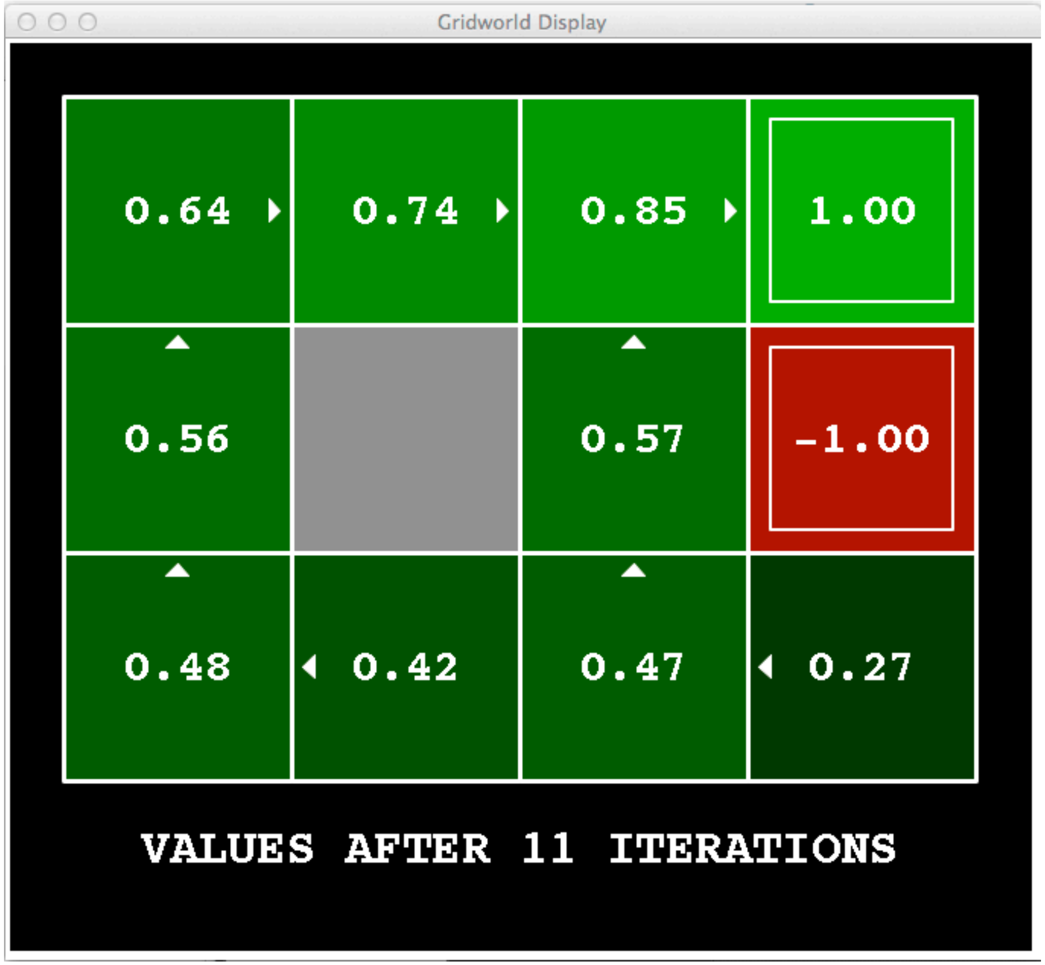


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.56		▴ 0.57	-1.00
▴ 0.48	◂ 0.41	▴ 0.47	◂ 0.27

VALUES AFTER 10 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=11

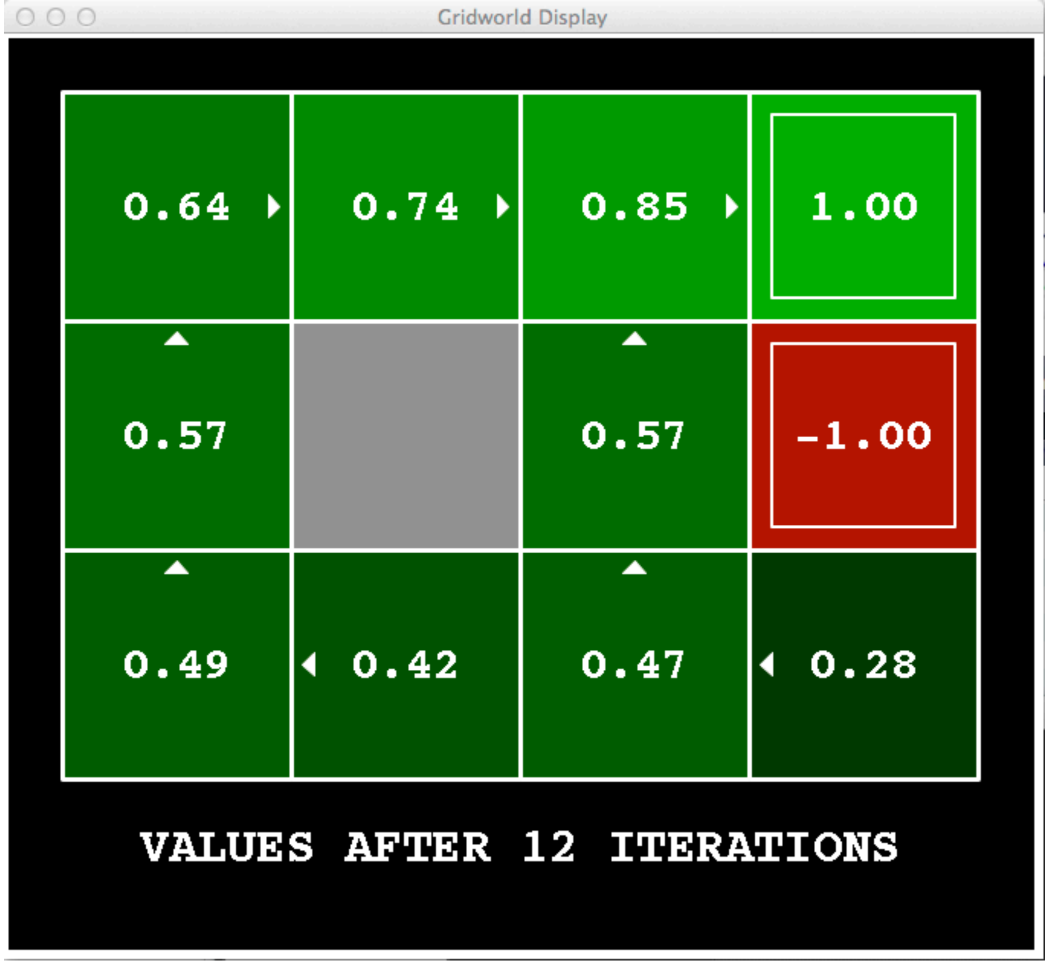


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.56		▴ 0.57	-1.00
▴ 0.48	◂ 0.42	▴ 0.47	◂ 0.27

VALUES AFTER 11 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=12

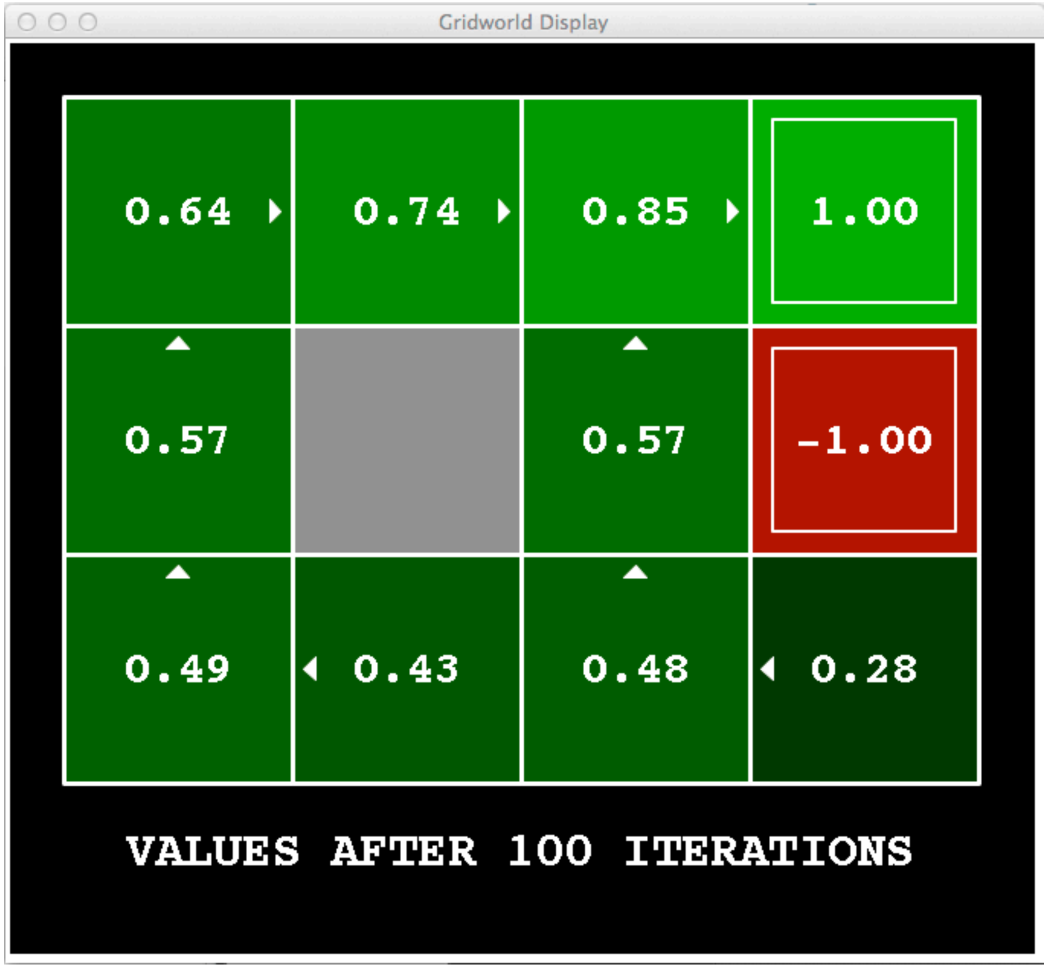


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.57		▴ 0.57	-1.00
▴ 0.49	◂ 0.42	▴ 0.47	◂ 0.28

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

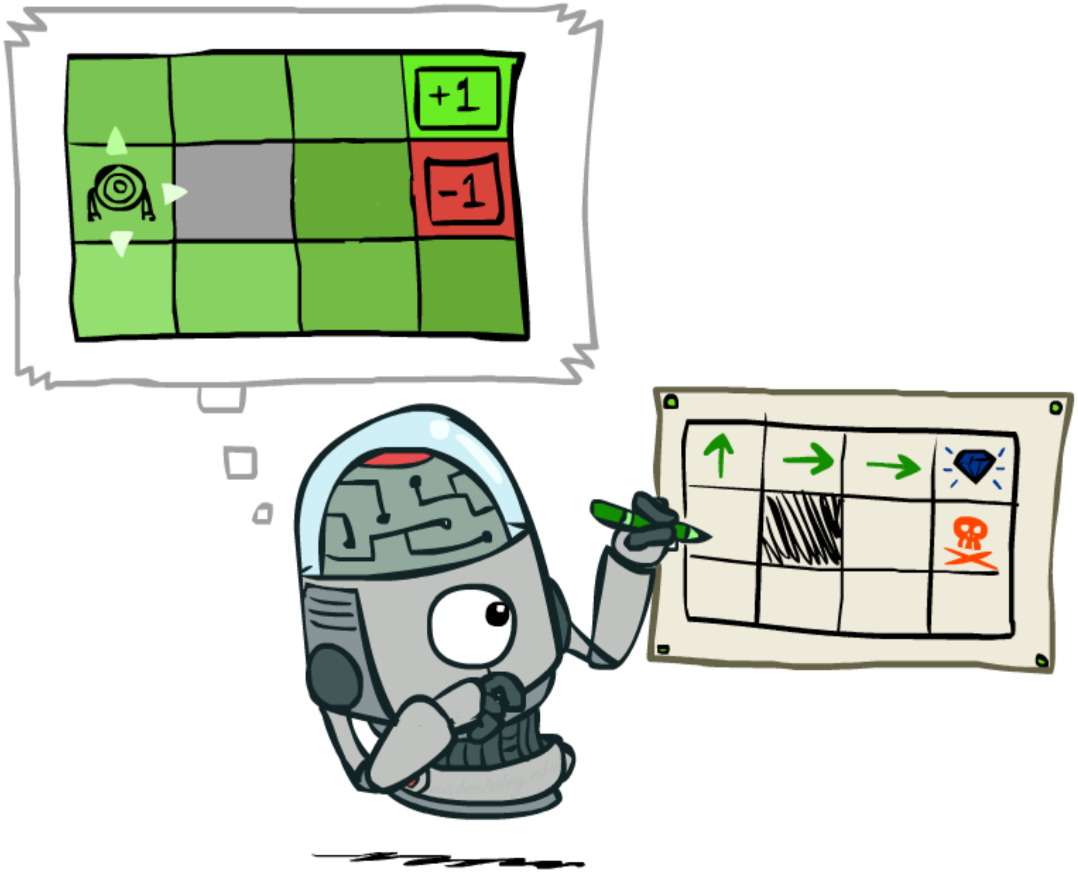


0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.57		▲ 0.57	-1.00
▲ 0.49	◀ 0.43	▲ 0.48	◀ 0.28

VALUES AFTER 100 ITERATIONS

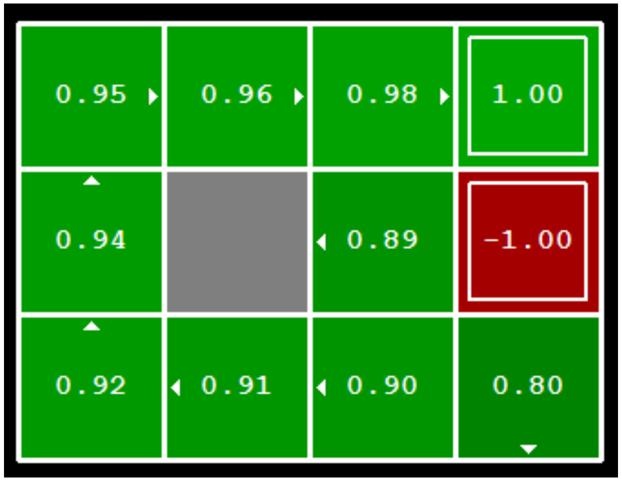
Noise = 0.2
Discount = 0.9
Living reward = 0

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)

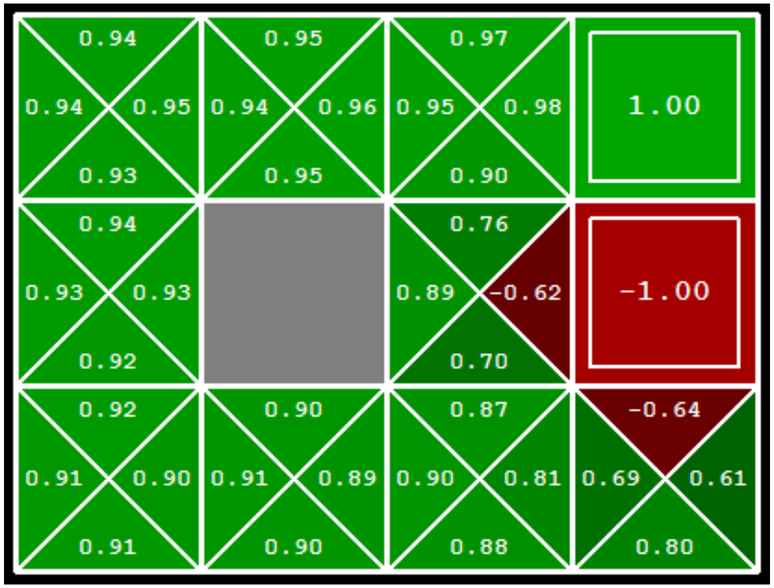


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- 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!

Value Iteration - Recap

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

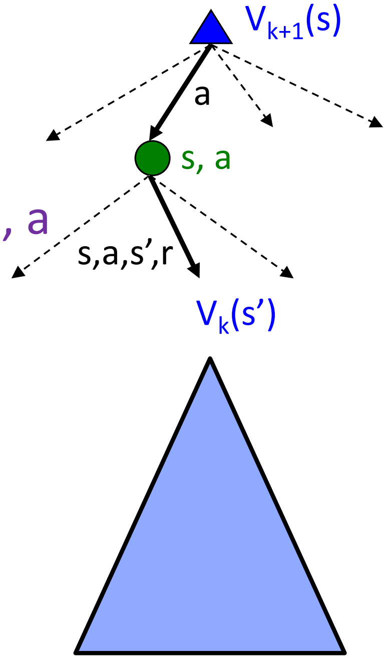
  K += 1

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

  $$\left.\begin{aligned} Q_{k+1}(s, a) &= \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \\ V_{k+1}(s) &= \text{Max}_{a} Q_{k+1}(s, a) \end{aligned}\right\}$$

  } do ∀ s, a

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

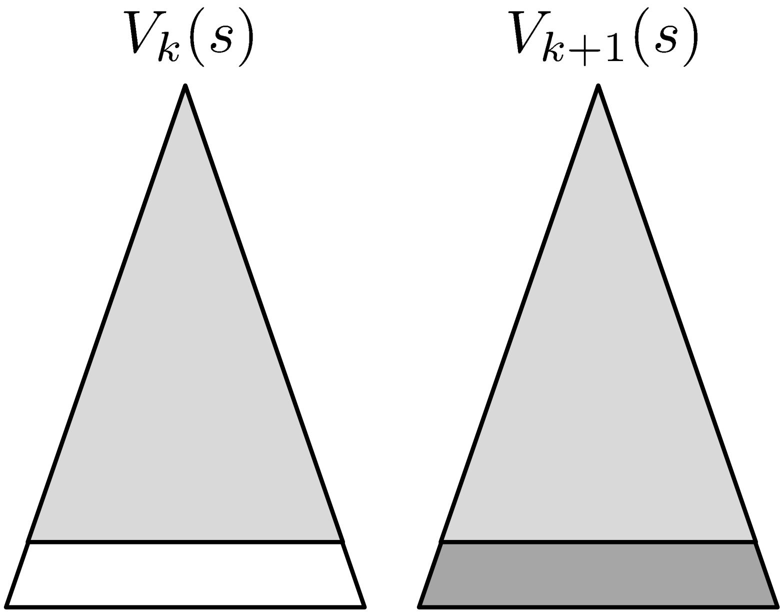


Convergence*

- 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

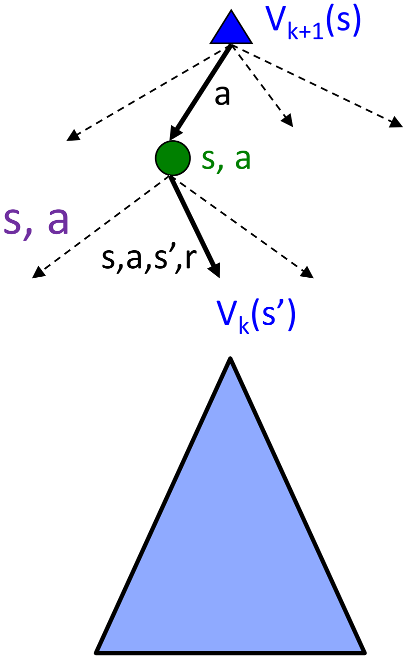
- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat** *do Bellman backups*

  K += 1

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

$$\left.\begin{aligned} Q_{k+1}(s, a) &= \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \\ V_{k+1}(s) &= \text{Max}_{a} Q_{k+1}(s, a) \end{aligned}\right\}$$

} do ∀ s, a



- **Until $|V_{k+1}(s) – V_k(s)| < \varepsilon$, forall s** *"convergence"*
- **Complexity of each iteration?**

Value Iteration - Recap

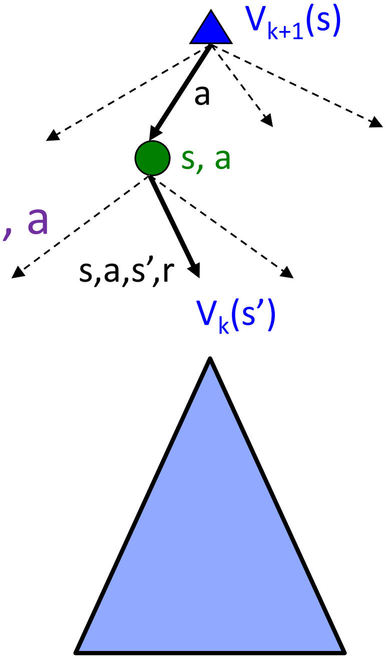
- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat** *do Bellman backups*

  K += 1

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

$$\left.\begin{aligned} Q_{k+1}(s, a) &= \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \\ V_{k+1}(s) &= \text{Max}_a\, Q_{k+1}(s, a) \end{aligned}\right\}$$

} do ∀ s, a



- **Until $|V_{k+1}(s) - V_k(s)| < \varepsilon$, forall s** *"convergence"*
- **Complexity of each iteration: $O(S^2A)$**
- **Number of iterations: poly(|S|, |A|, 1/(1-γ))**

Value Iteration as Successive Approximation

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

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

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

- Value iteration ***computes*** them:

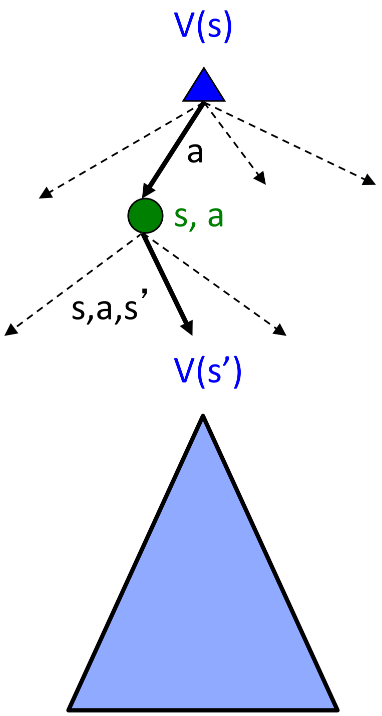
$$Q_{k+1}(s, a) = \Sigma_{s'}\ T(s, a, s')\ [\ R(s, a, s') + \gamma\ V_k(s')]$$

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

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

Computed using dynamic programming

… though the V_k vectors are also interpretable as time-limited values

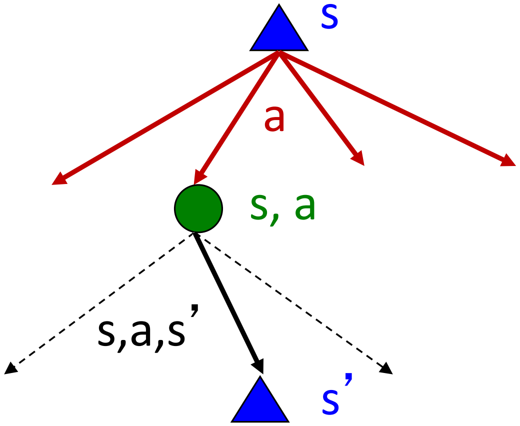


Problems with Value Iteration

- 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

VI → Asynchronous VI

- Is it essential to back up ***all*** states in each iteration?
  - No!

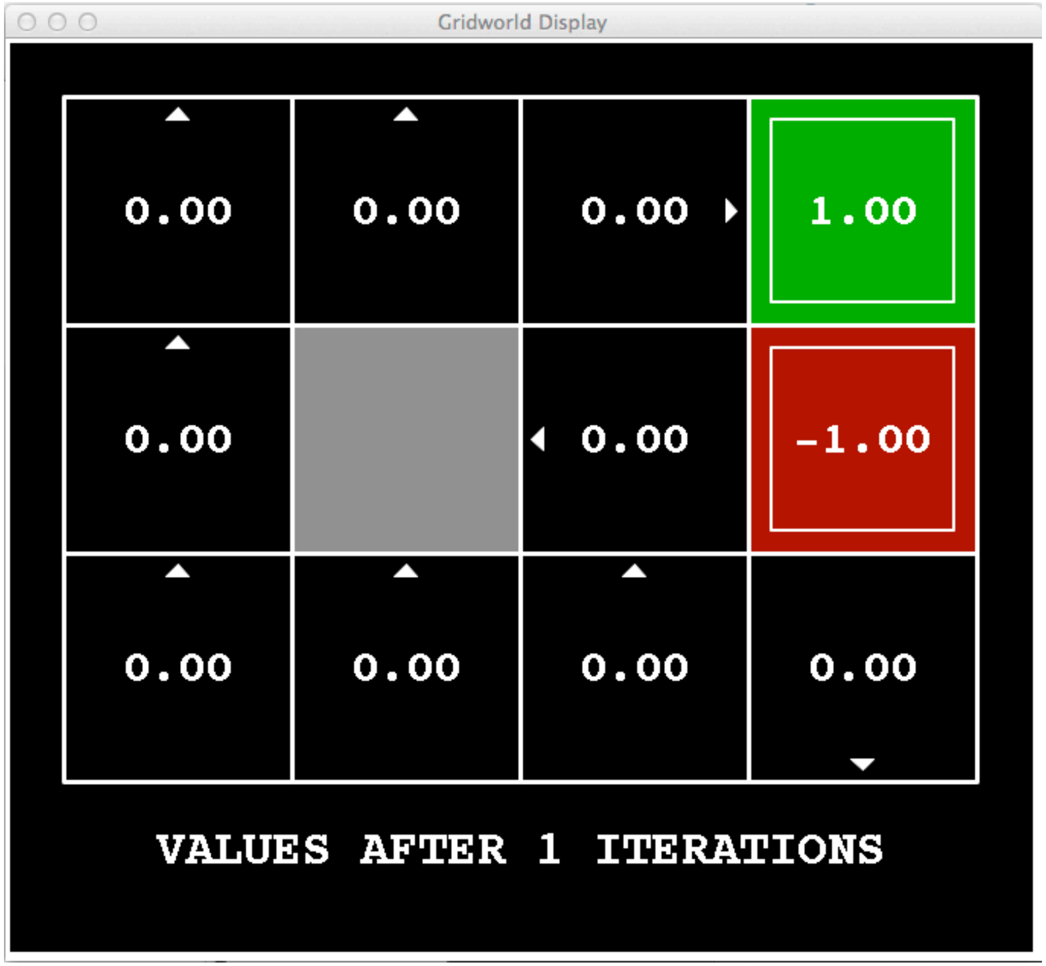
- States may be backed up
  - many times or not at all
  - in any order

- As long as no state gets starved…
  - convergence properties still hold!!

Prioritization of Bellman Backups

- Are all backups equally important?
- Can we avoid some backups?
- Can we schedule the backups more appropriately?

k=1

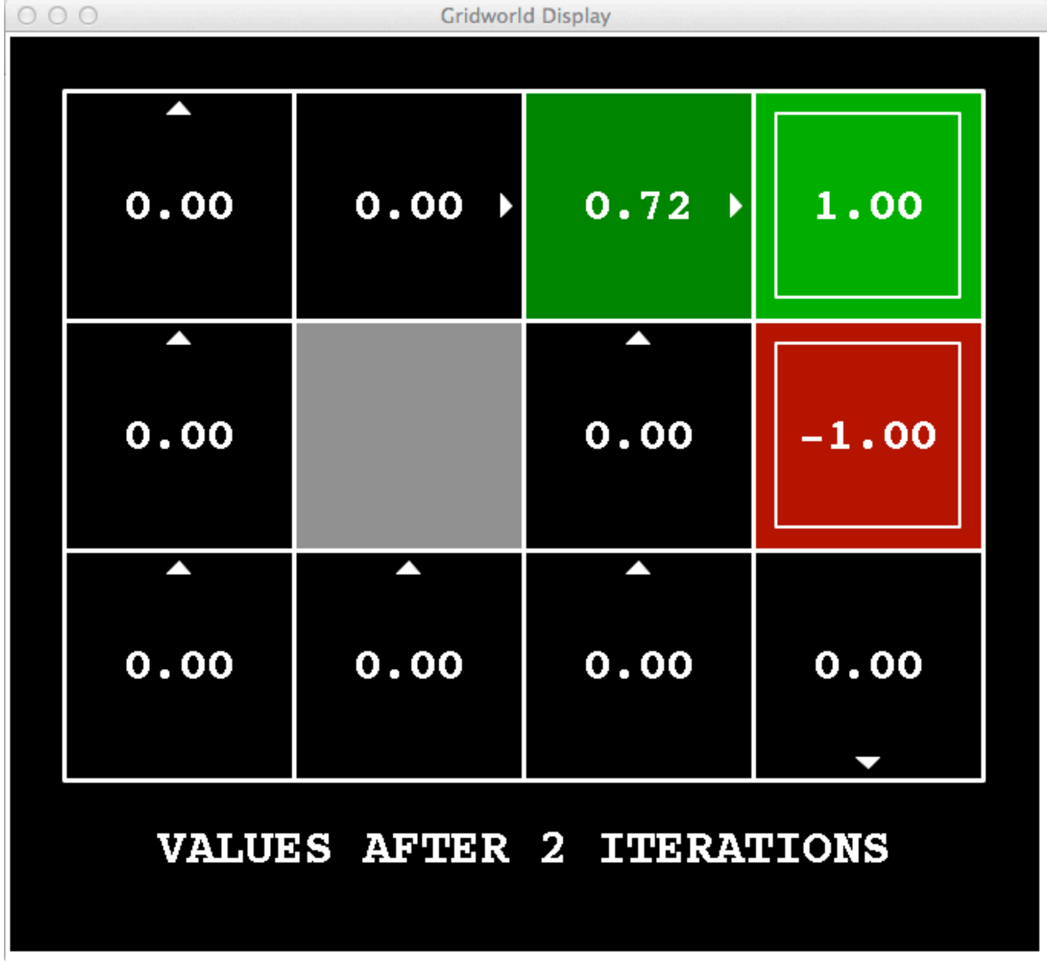


▲ 0.00	▲ 0.00	0.00 ▶	1.00
▲ 0.00		◀ 0.00	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 1 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

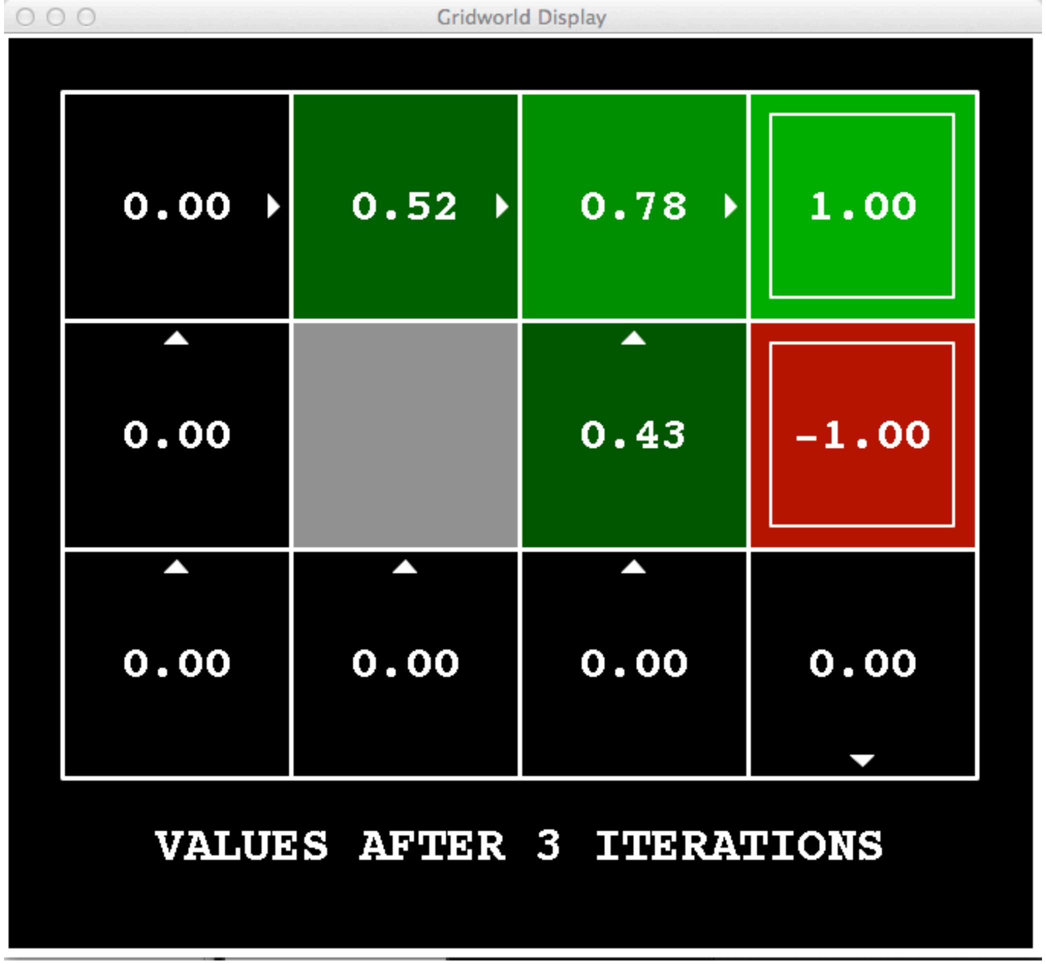


▲ 0.00	0.00 ▶	0.72 ▶	1.00
▲ 0.00		▲ 0.00	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 2 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

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

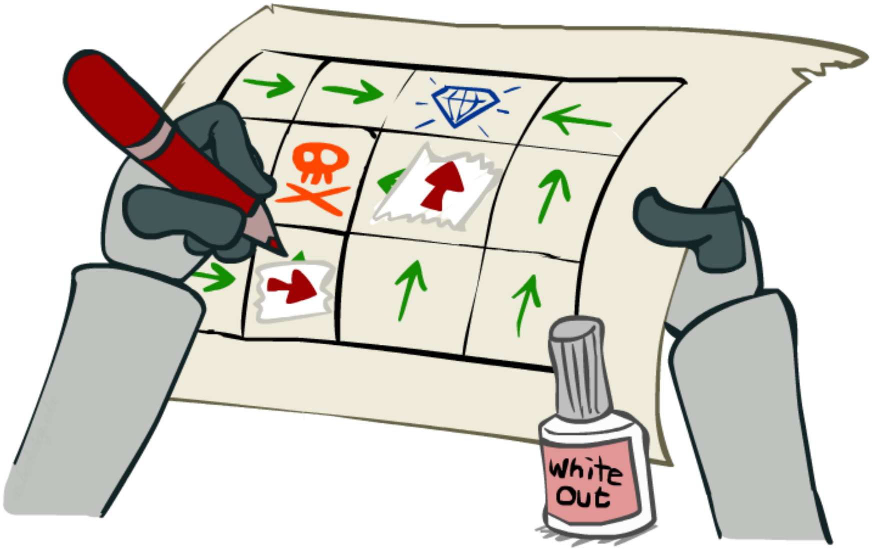
Noise = 0.2
Discount = 0.9
Living reward = 0

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 ~ residual)
- Residual at *s* with respect to *V*
  - magnitude(Δ*V(s))* after one Bellman backup at s

$$\mathrm{Res}_v(s) = |\ V(s) - \max_{a \in A} \sum_{s' \in S} T(s,a,s')[R(s,a,s')+V(s')]\ |$$

Solving MDPs



- Value Iteration
- Policy Iteration
- Heuristic Search Methods
- Real-Time Dynamic programming
- Reinforcement Learning