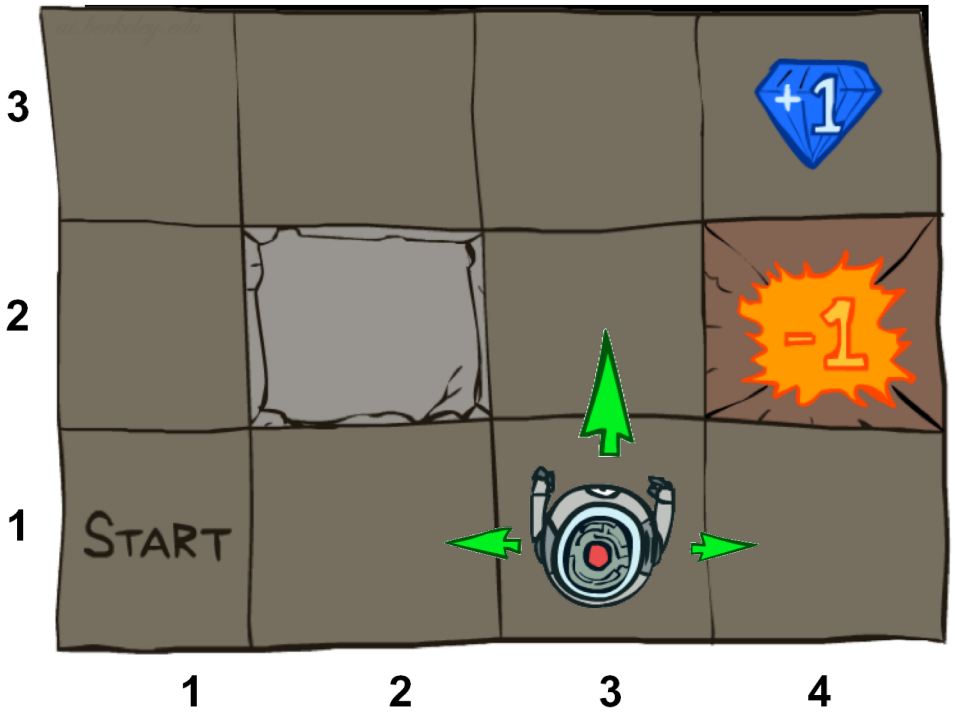
CSC 665: Artificial Intelligence

Markov Decision Processes

Instructor: Pooyan Fazli
San Francisco State University

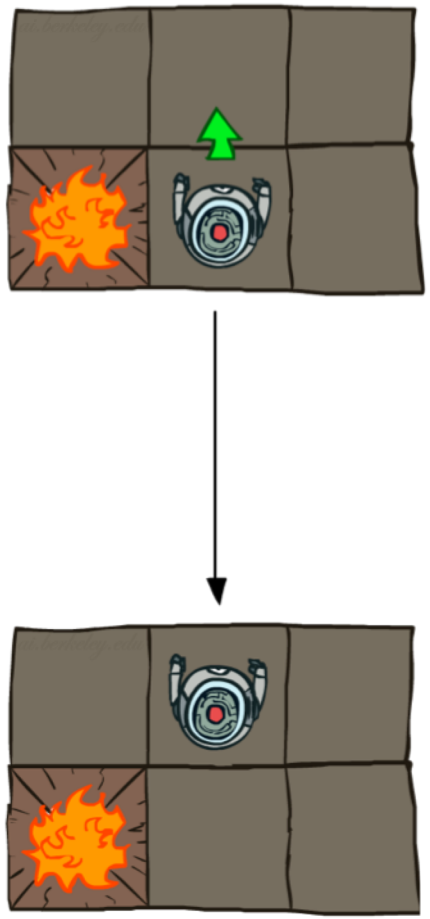
Example: Grid World

- 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

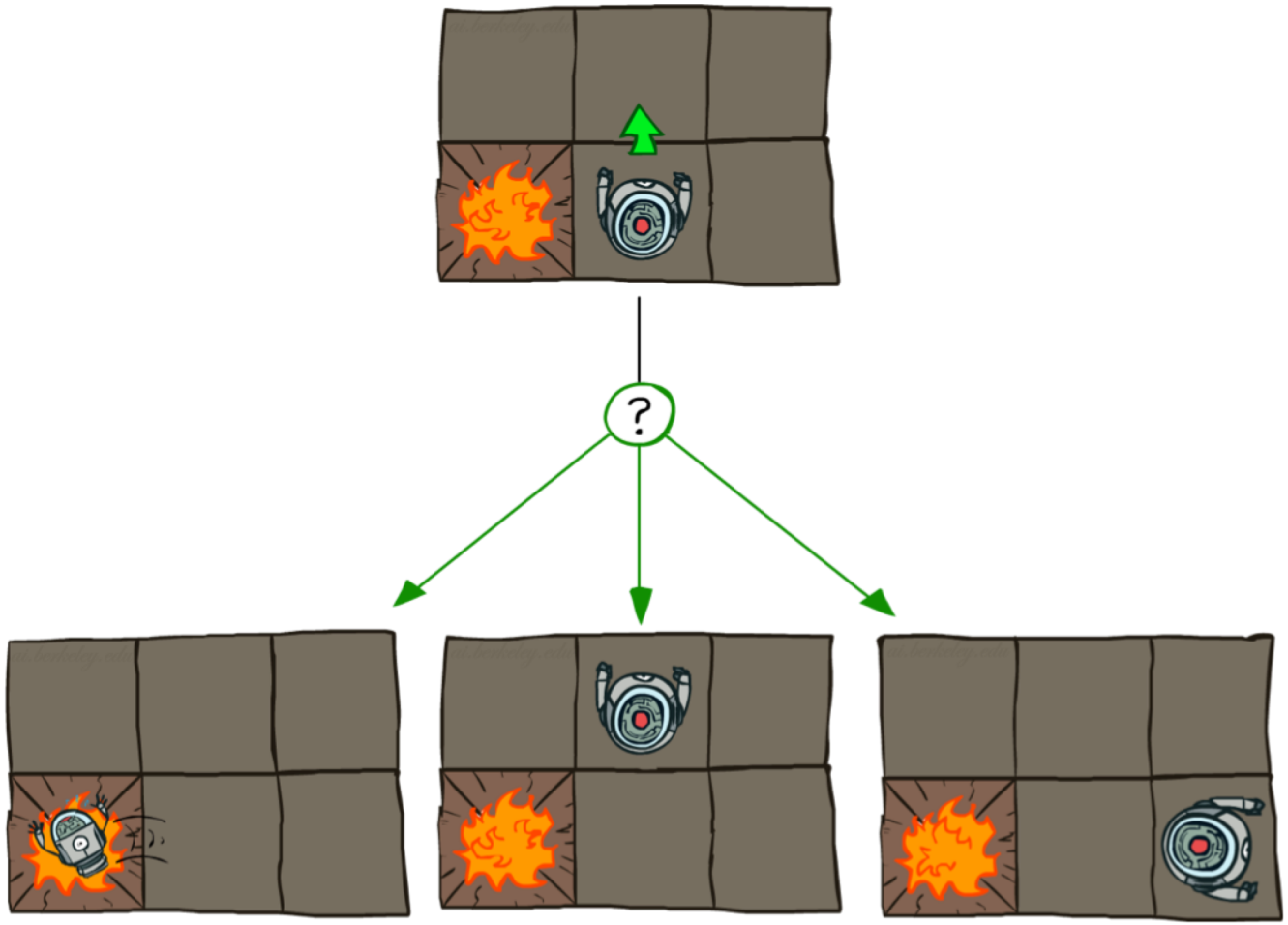


Grid World Actions

Deterministic Grid World

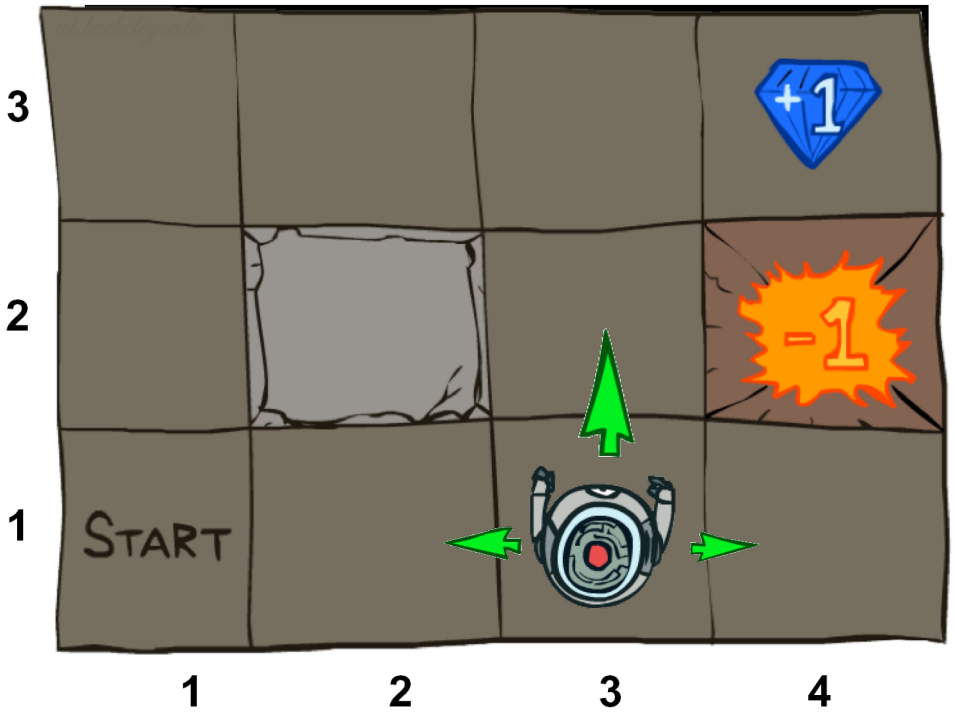


Stochastic Grid World



Markov Decision Processes

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

What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent

- For Markov decision processes, "Markov" means the next state depends only on the current state and action

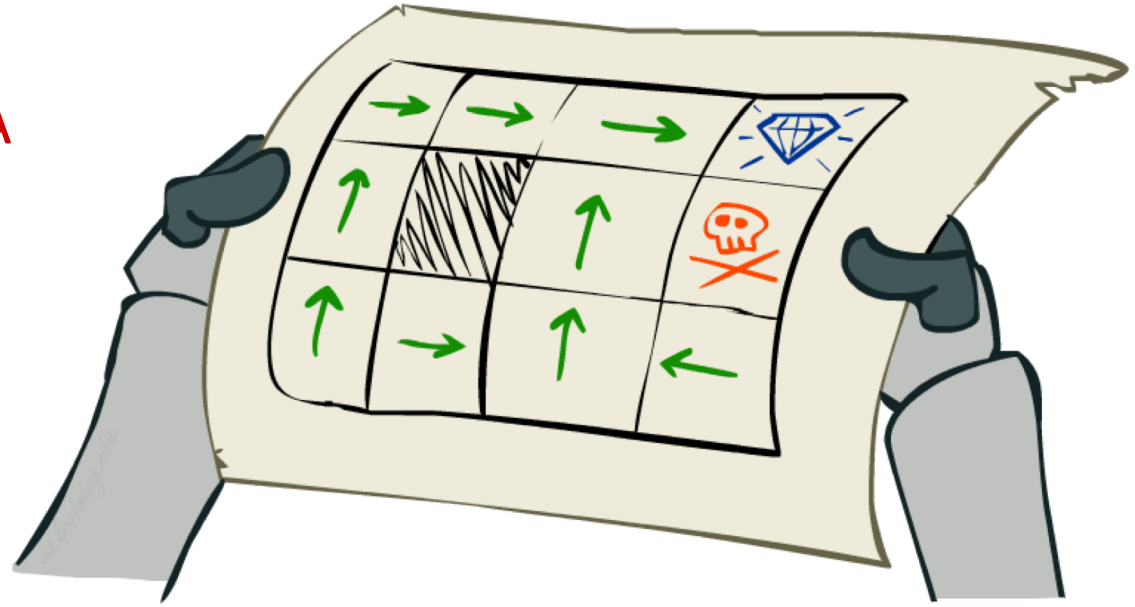
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$
$$=$$
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

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

Andrey Markov
(1856-1922)

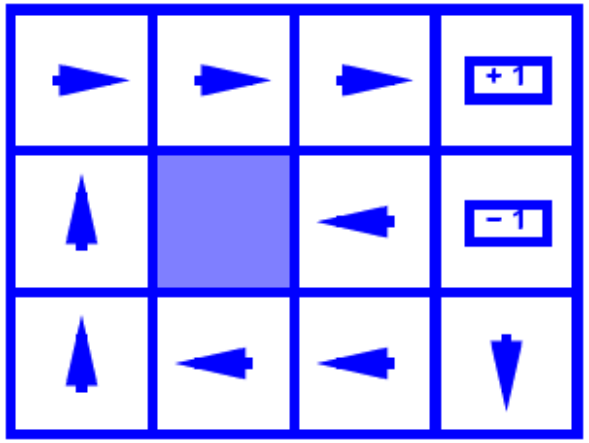
Policies

- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy π^* is one that maximizes expected utility if followed

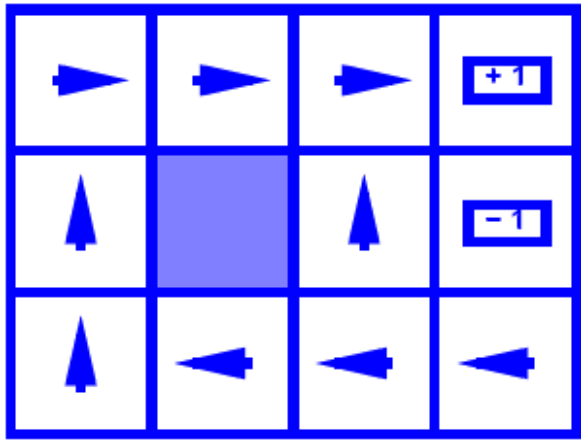


Optimal policy when R(s, a, s') = -0.03
for all non-terminals s

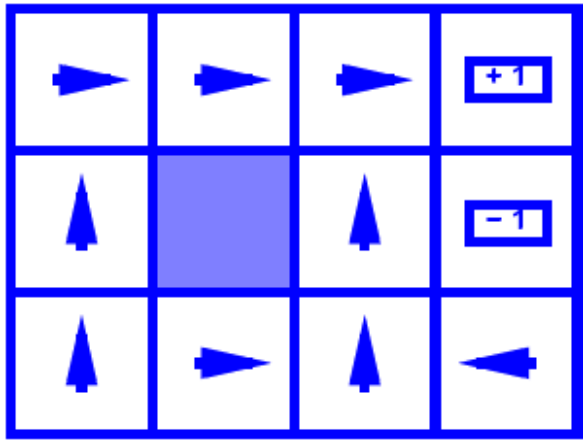
Optimal Policies



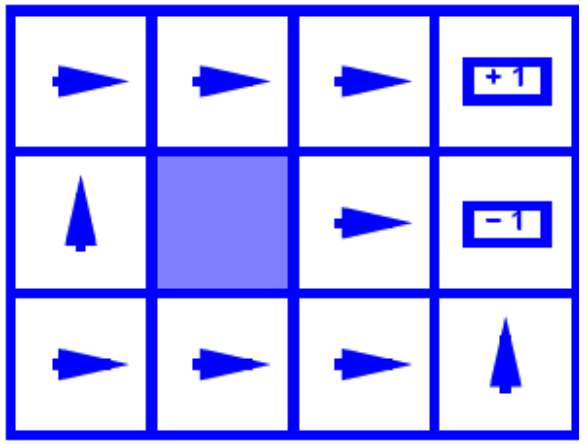
R(s, a, s') = -0.01



R(s, a, s') = -0.08



R(s, a, s') = -0.4

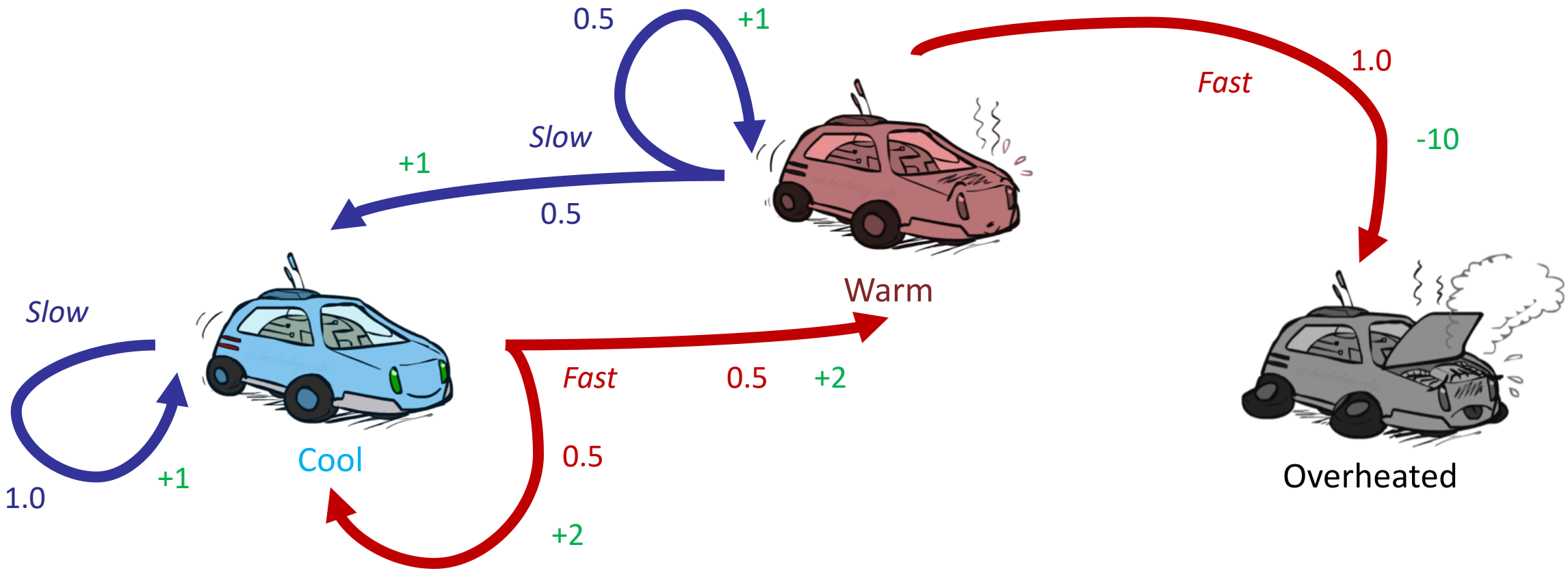


R(s, a, s') = -2.0

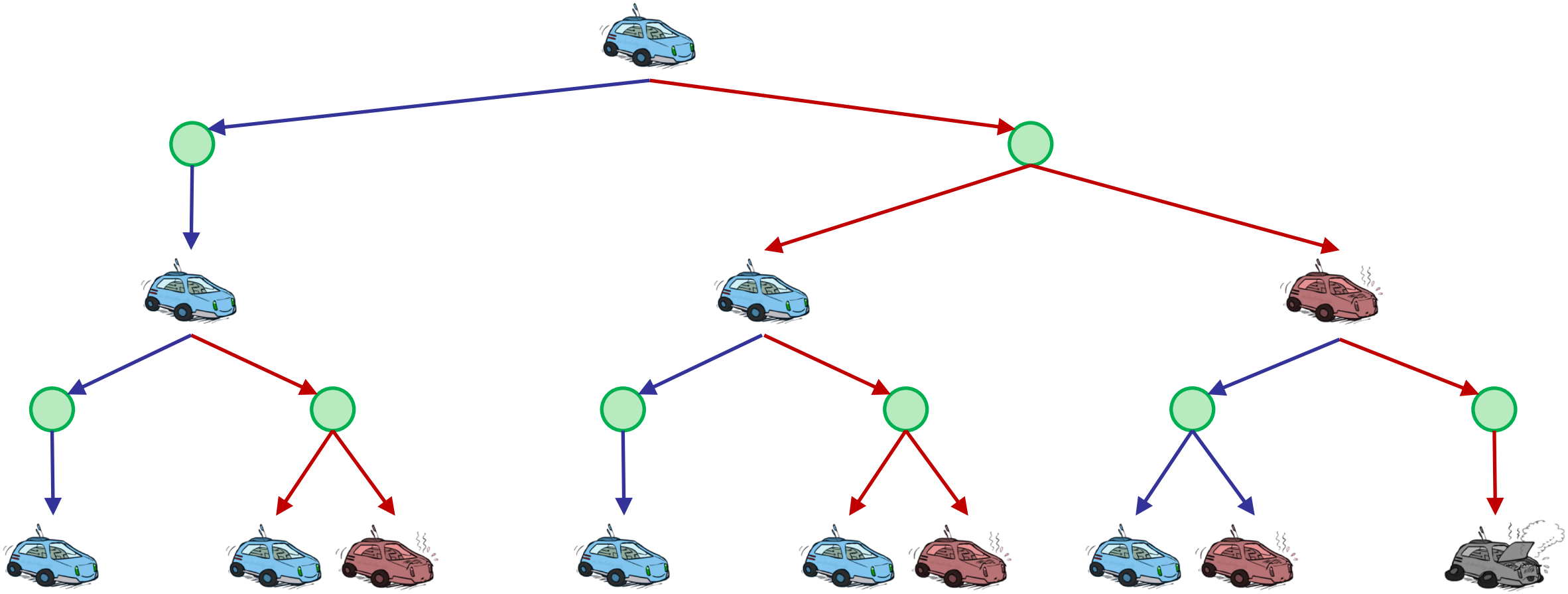
Example: Racing

Example: Racing

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: *Slow*, *Fast*

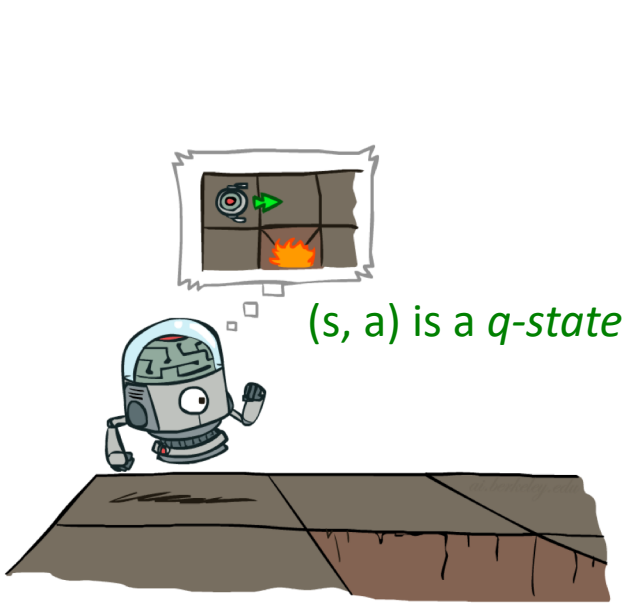


Racing Search Tree

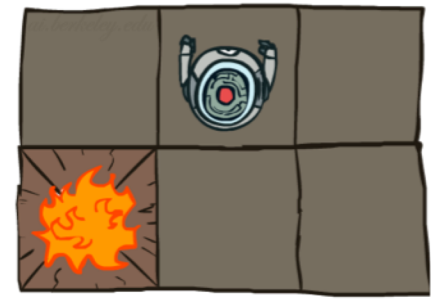


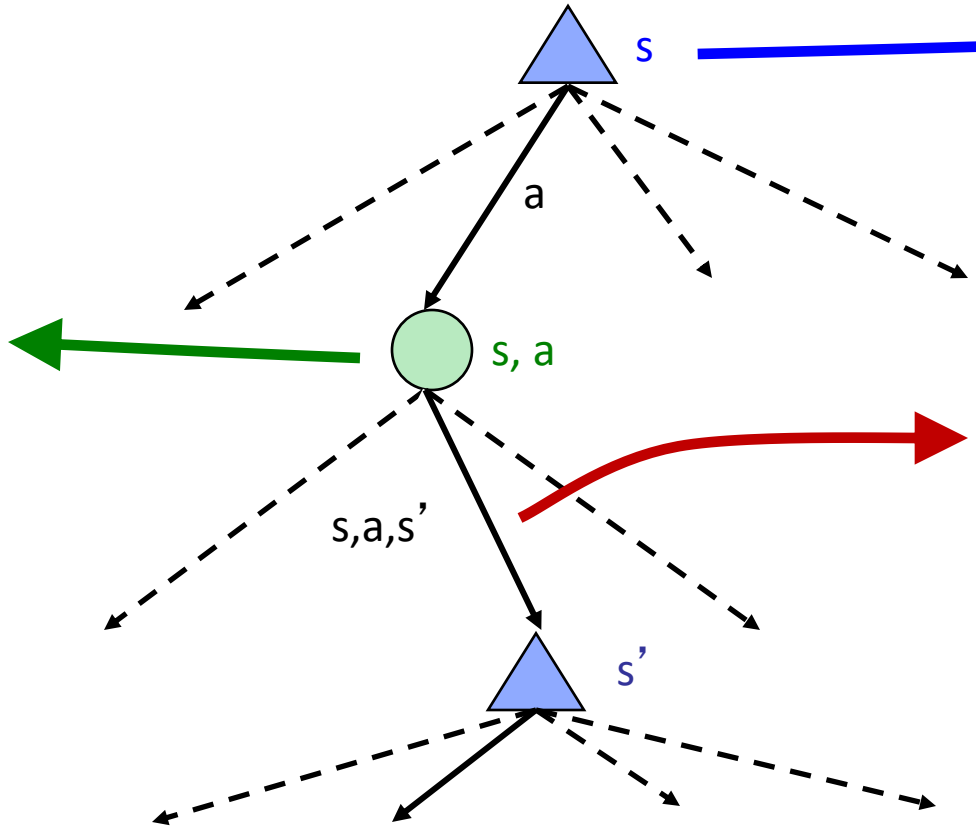
MDP Search Trees

- Each MDP state projects an expectimax-like search tree



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





s is a *state*

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

T(s,a,s') = P(s'|s,a)

R(s,a,s')

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]

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 $0<\gamma<1$

1

Worth Now

γ

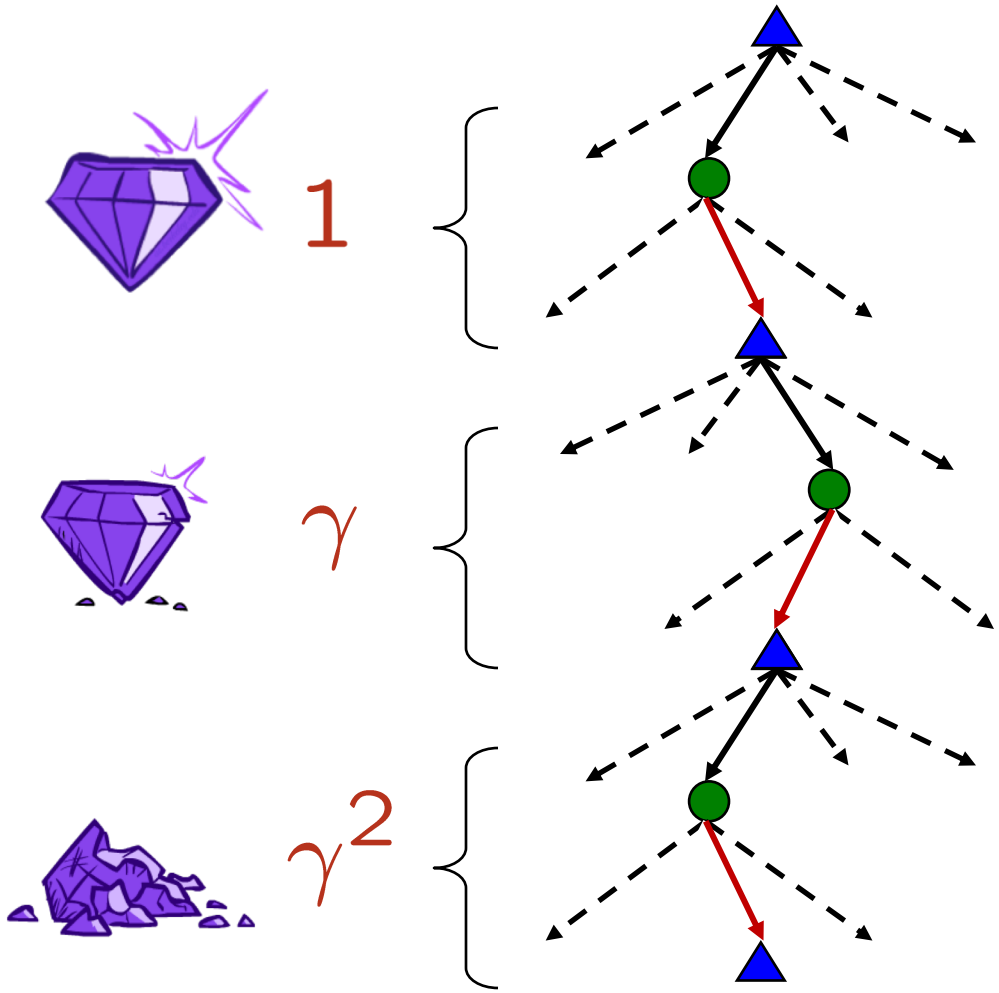
Worth Next Step

γ^2

Worth In Two Steps

Discounting

- How to discount?
  - Each time we descend a level, we multiply 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])



Stationary Preferences

- Theorem: if we assume stationary preferences:

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

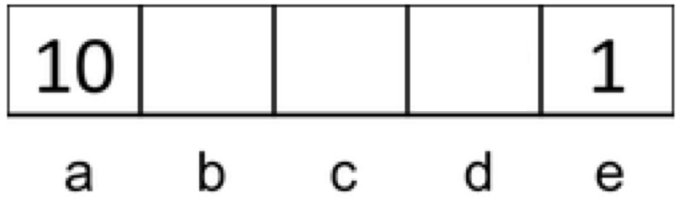
$$\Updownarrow$$

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

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

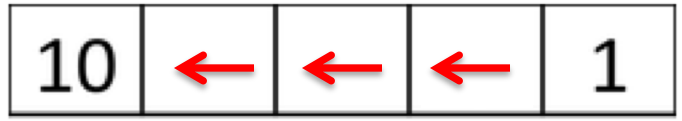
Quiz: Discounting

- Given:

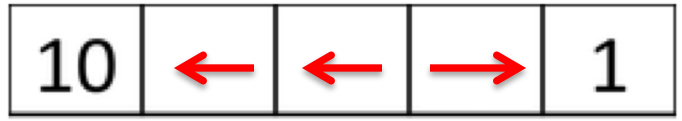


  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

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



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



- Quiz 3: For which γ are West and East equally good when in state d?

Sqrt(1/10)

Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives *nonstationary* policies (i.e. π depends on time left)
  - Discounting: use $0 < \gamma < 1$

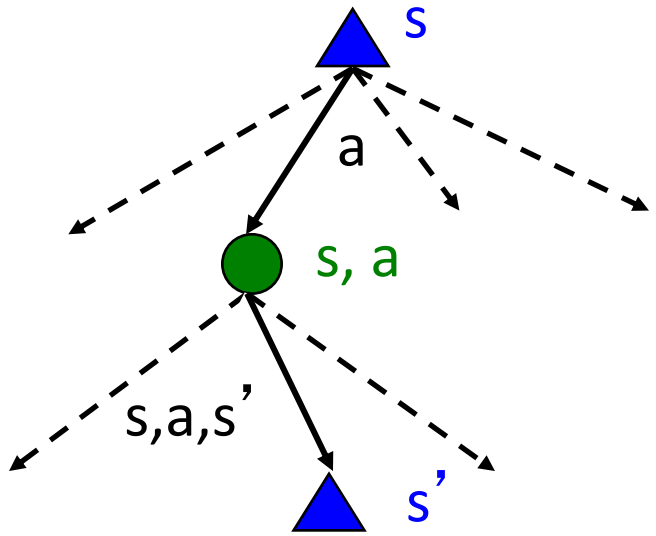
$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1-\gamma)$$

$$S_n = \frac{a_0(1-r^n)}{1-r}$$

    - Smaller γ means smaller "horizon" – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)

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

Solving MDPs

- We have an MDP Problem
- Compute the V* and Q* values for the states
- Extract the optimal action for each state (optimal policy)

Optimal Quantities

- The value (utility) of a state s:

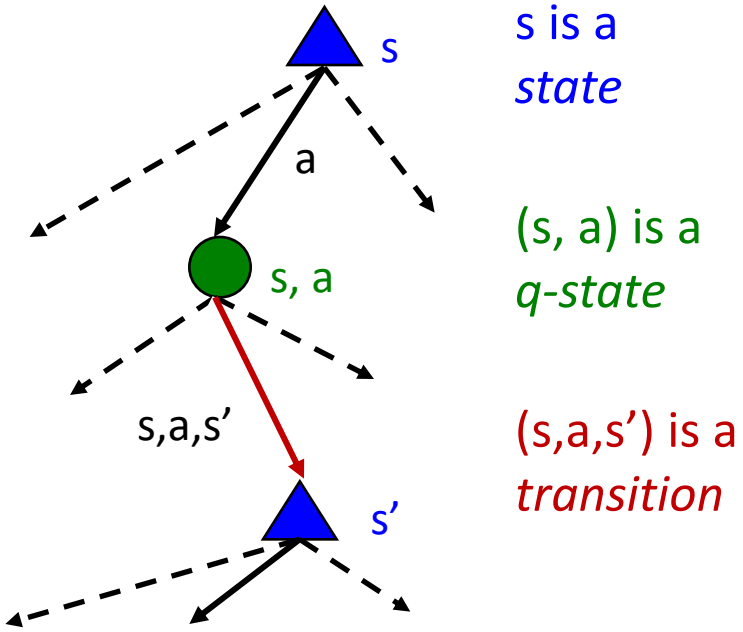
  $V^*(s)$ = expected utility starting in s and following the optimal policy

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

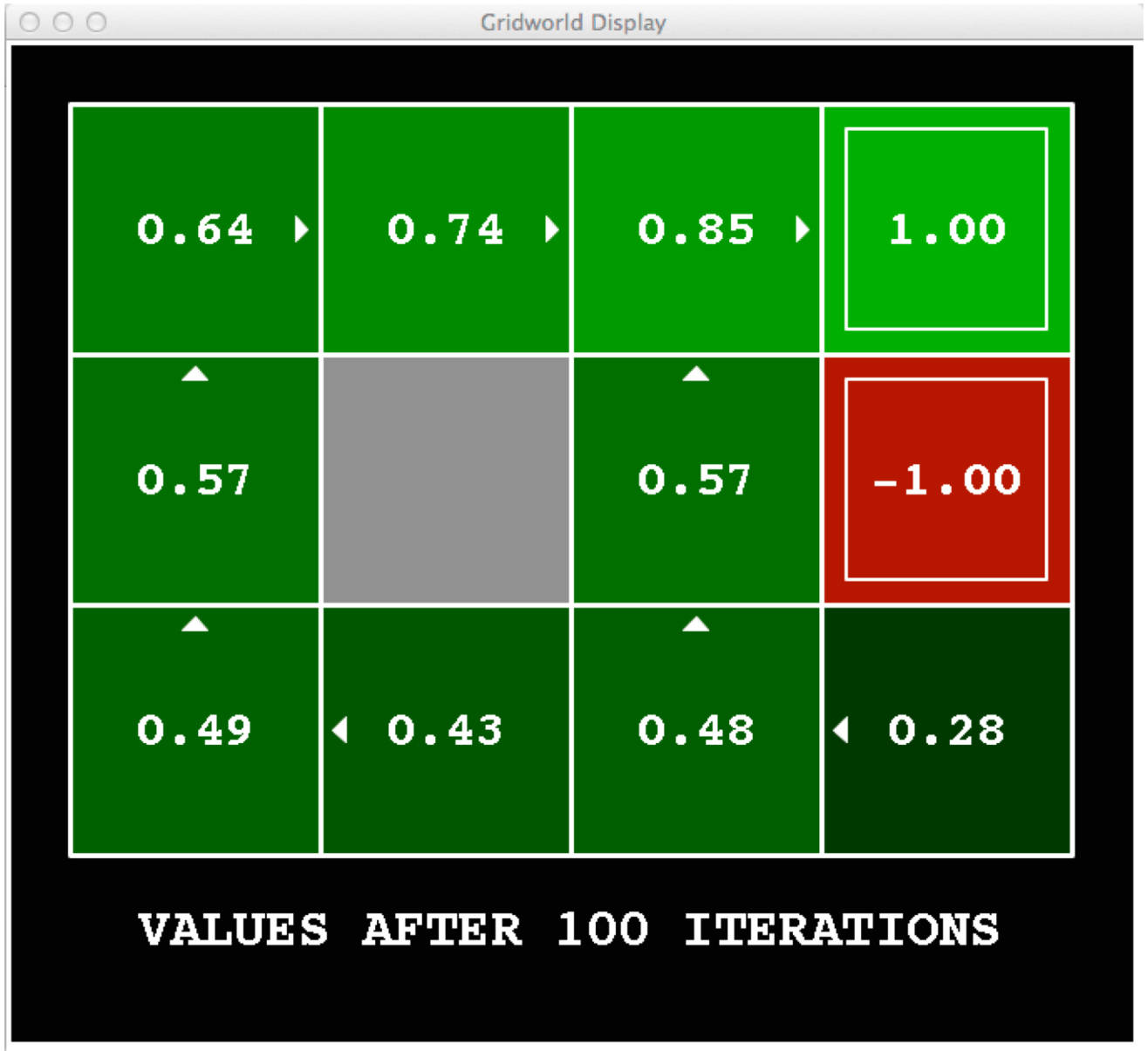
  $Q^*(s,a)$ = expected utility of doing a in state s and then following the optimal policy

- The optimal policy:

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

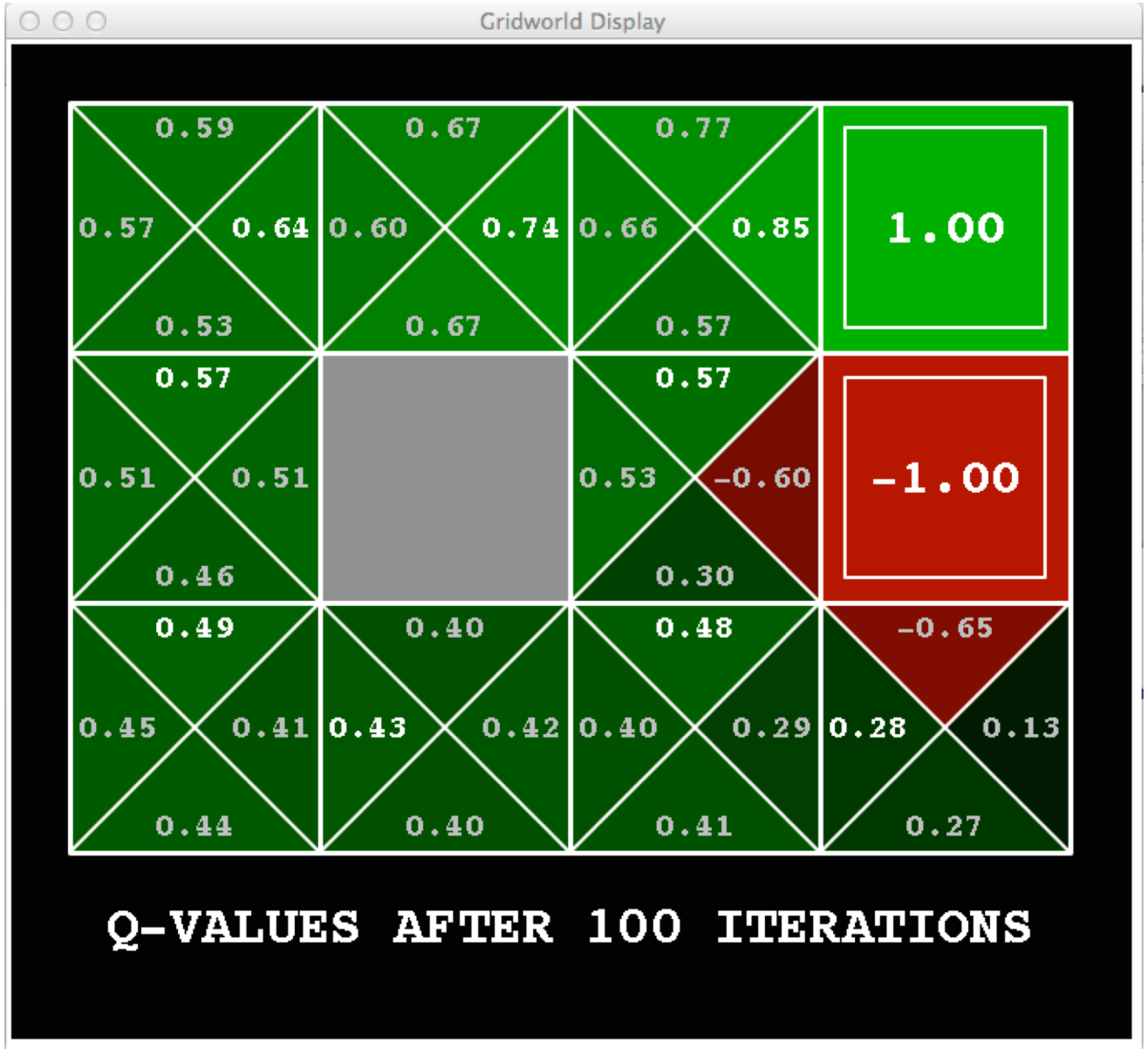


Gridworld V* Values



Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld Q* Values

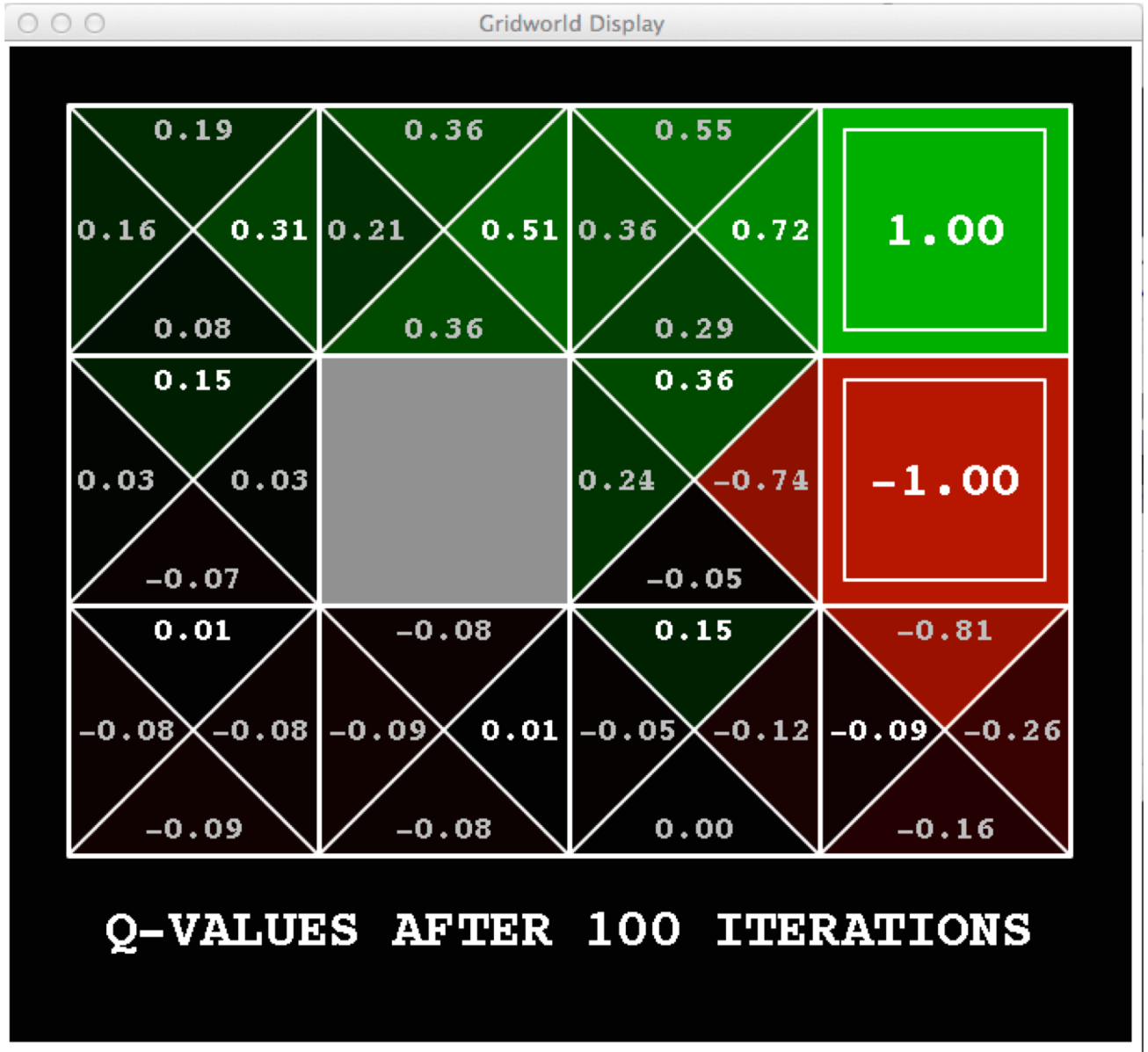


Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld V* Values

Noise = 0.2
Discount = 0.9
Living reward = -0.1

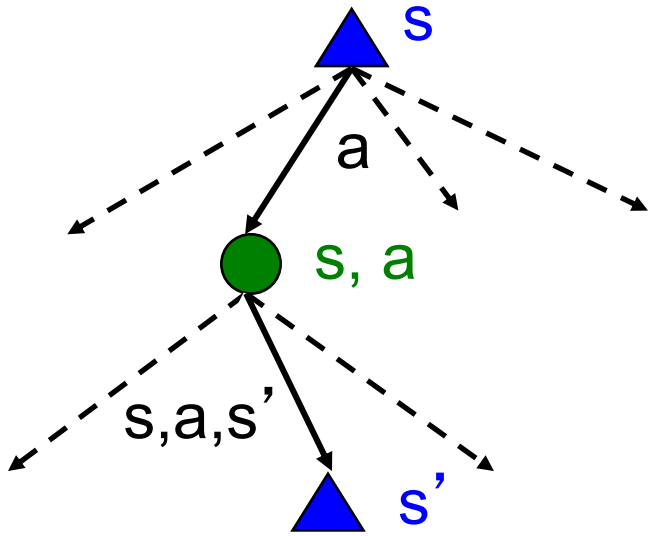
Gridworld Q* Values



Noise = 0.2
Discount = 0.9
Living reward = -0.1

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!



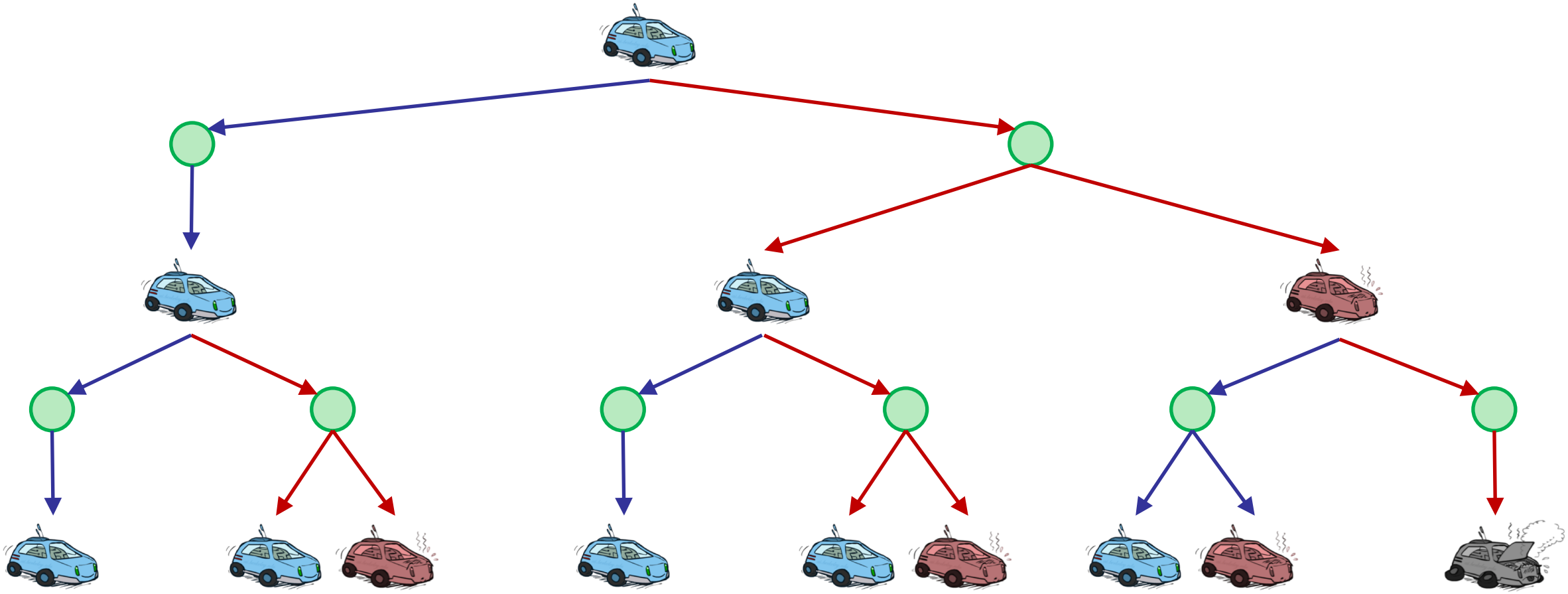
- Recursive definition of value:

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

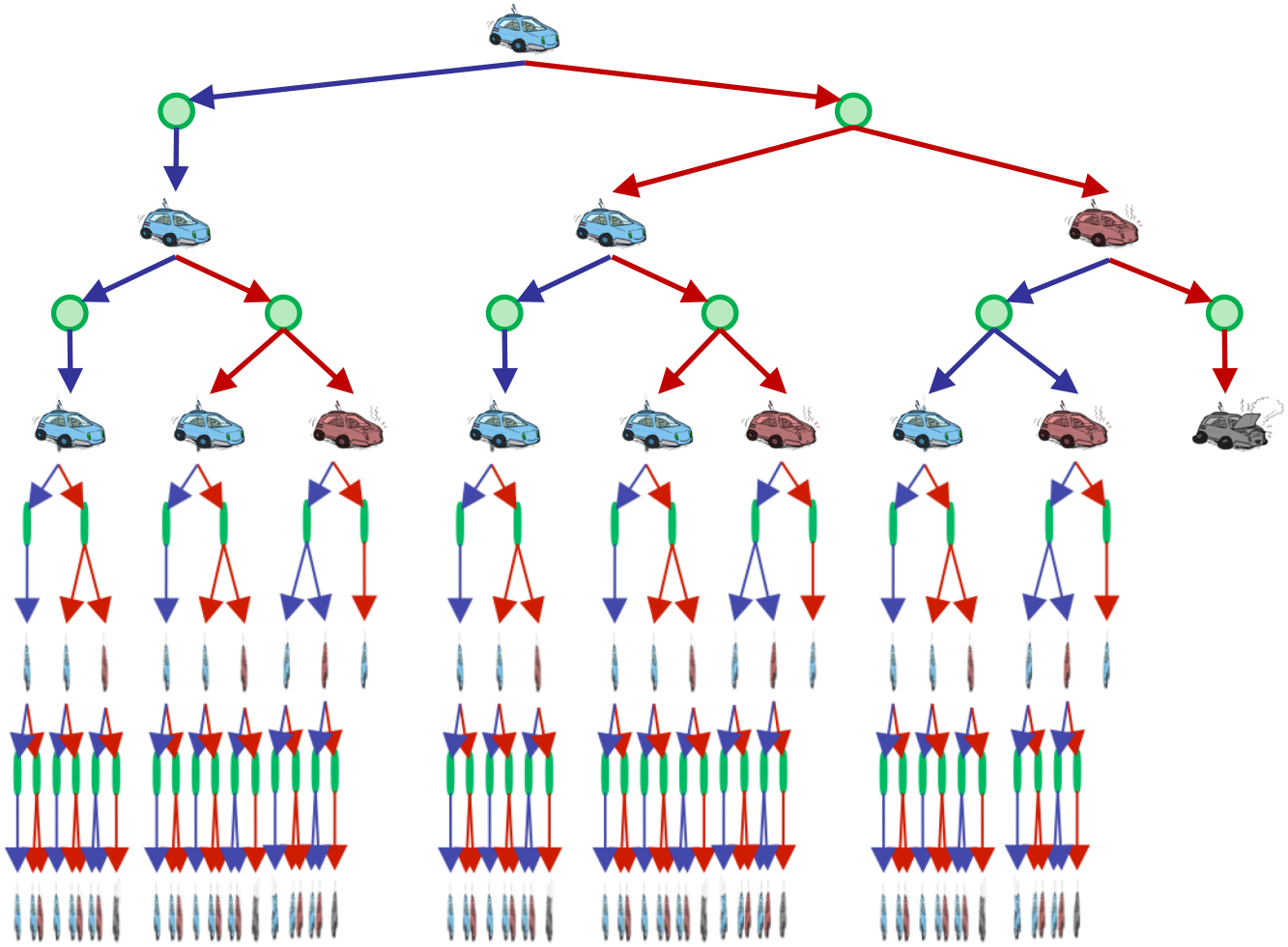
$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

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

Racing Search Tree

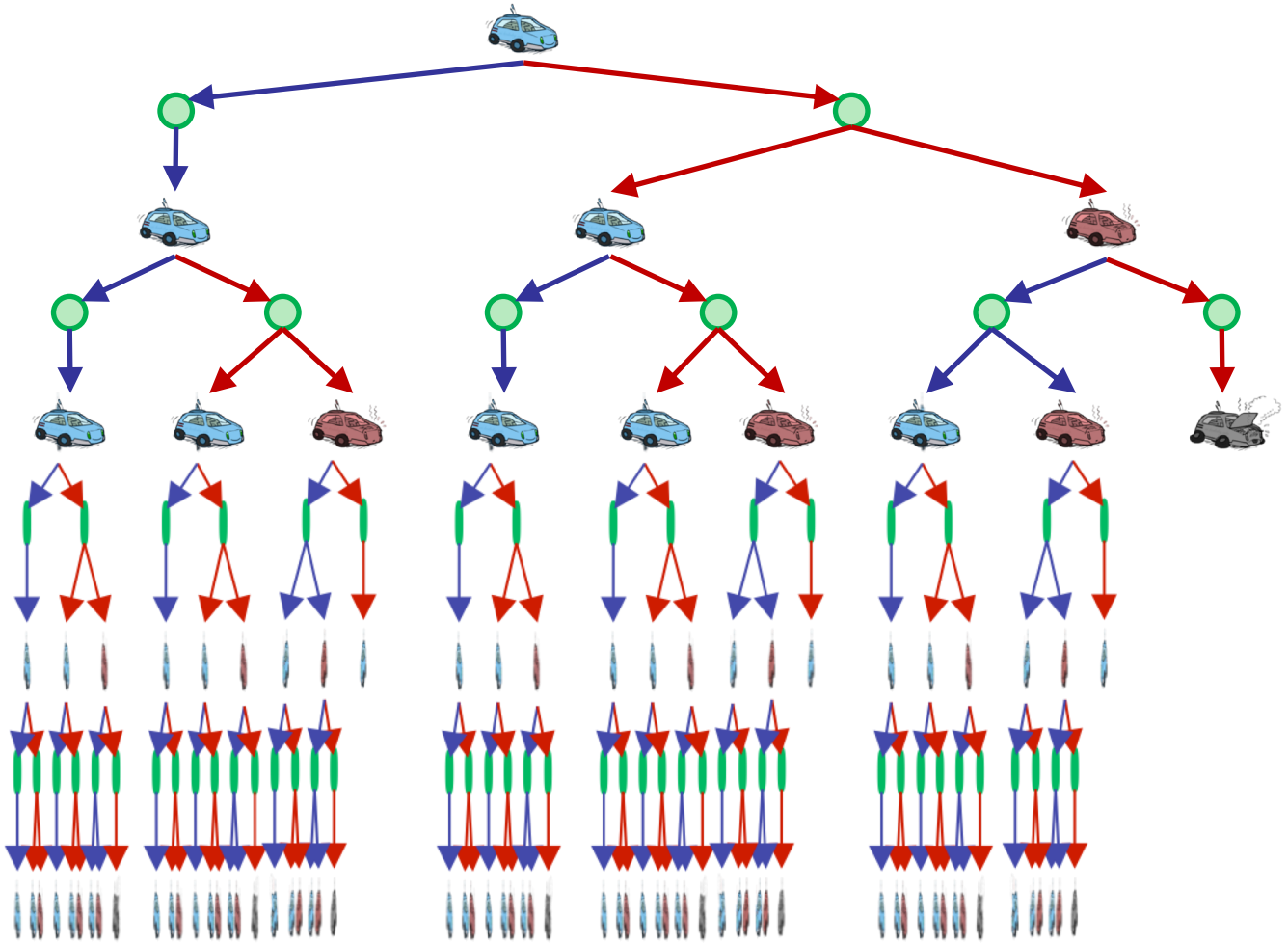


Racing Search Tree



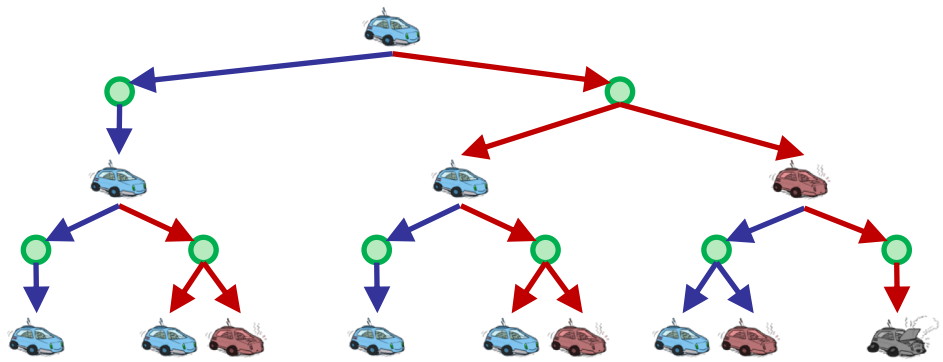
Racing Search Tree

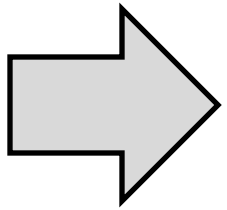
- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter if $\gamma < 1$

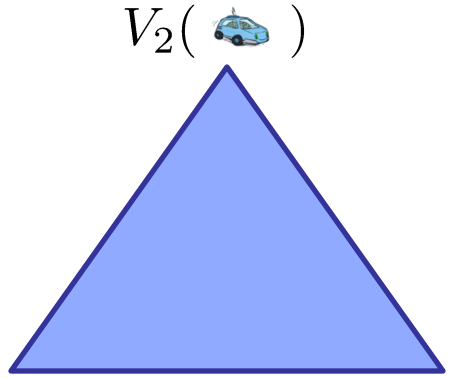


Time-Limited Values

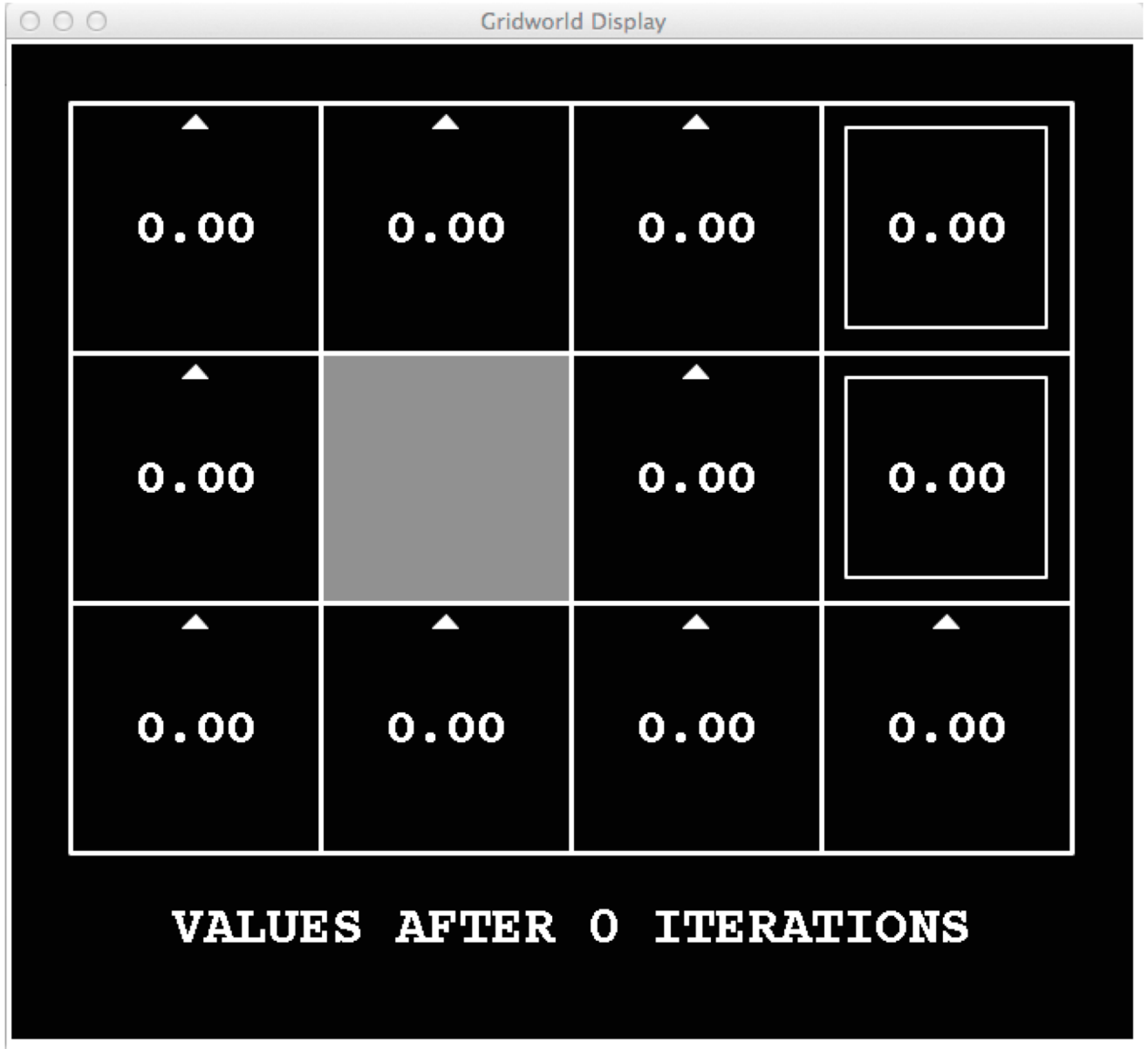
- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
  - Equivalently, it's what a depth-k expectimax would give from s





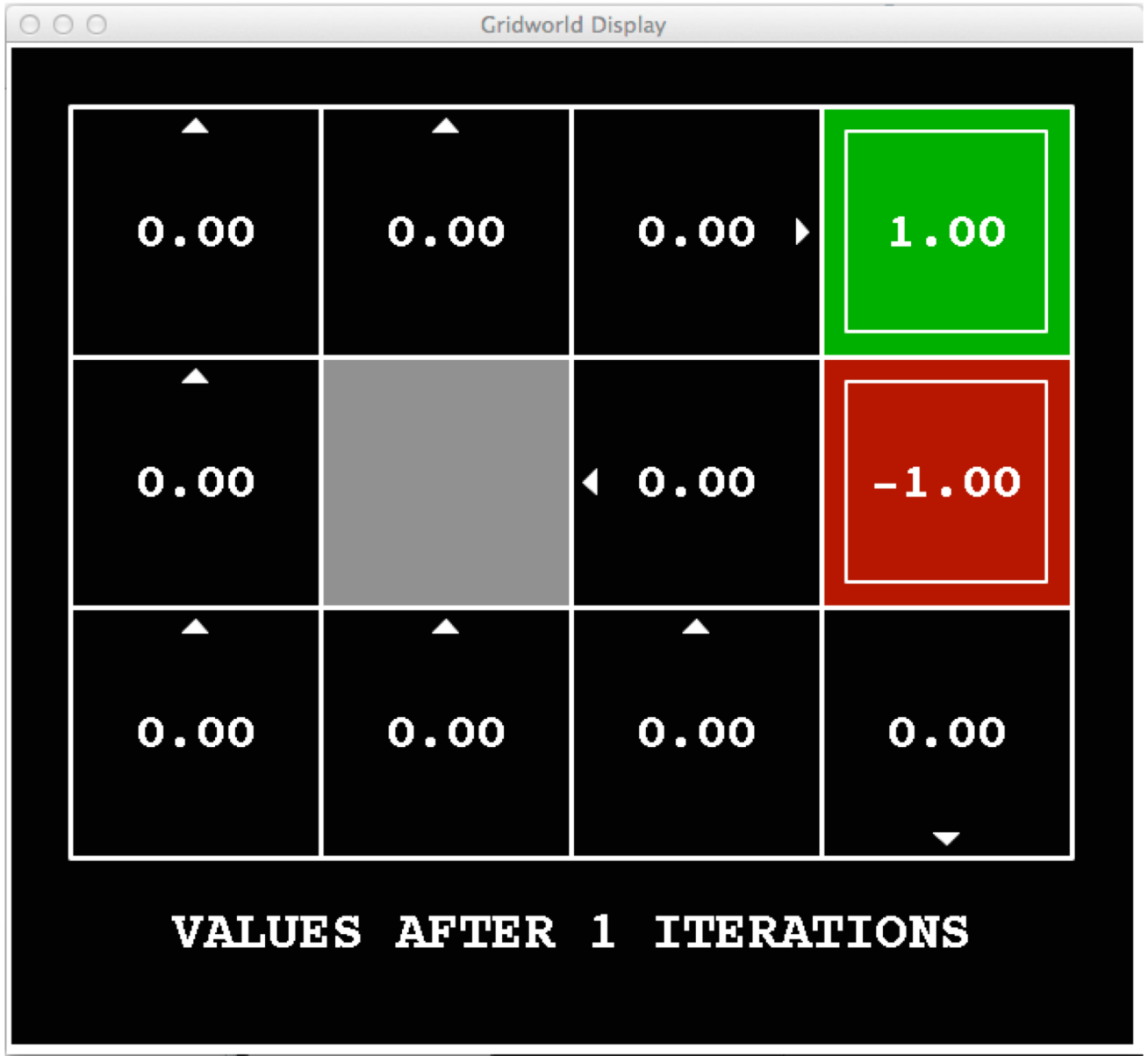


k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

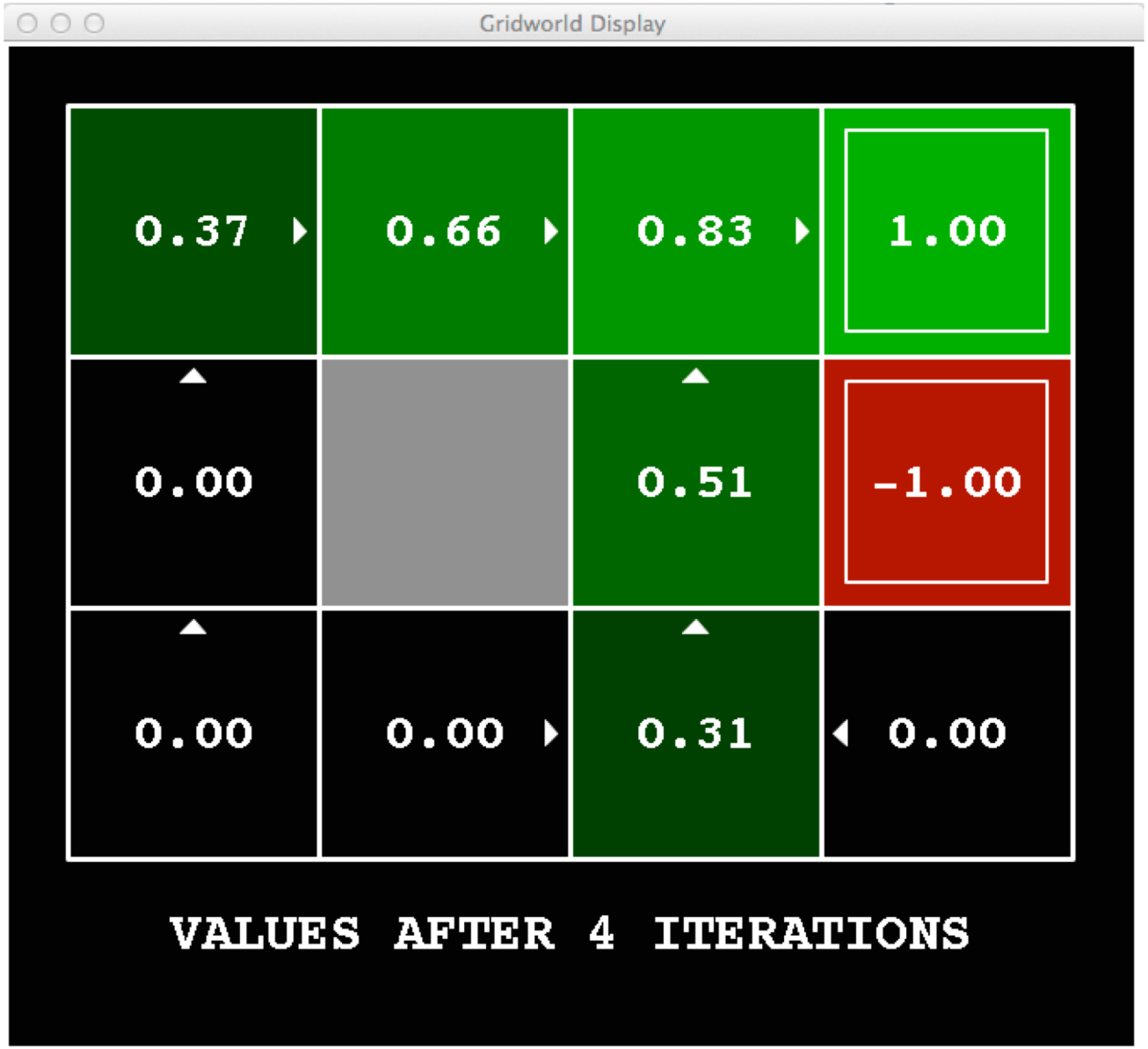
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

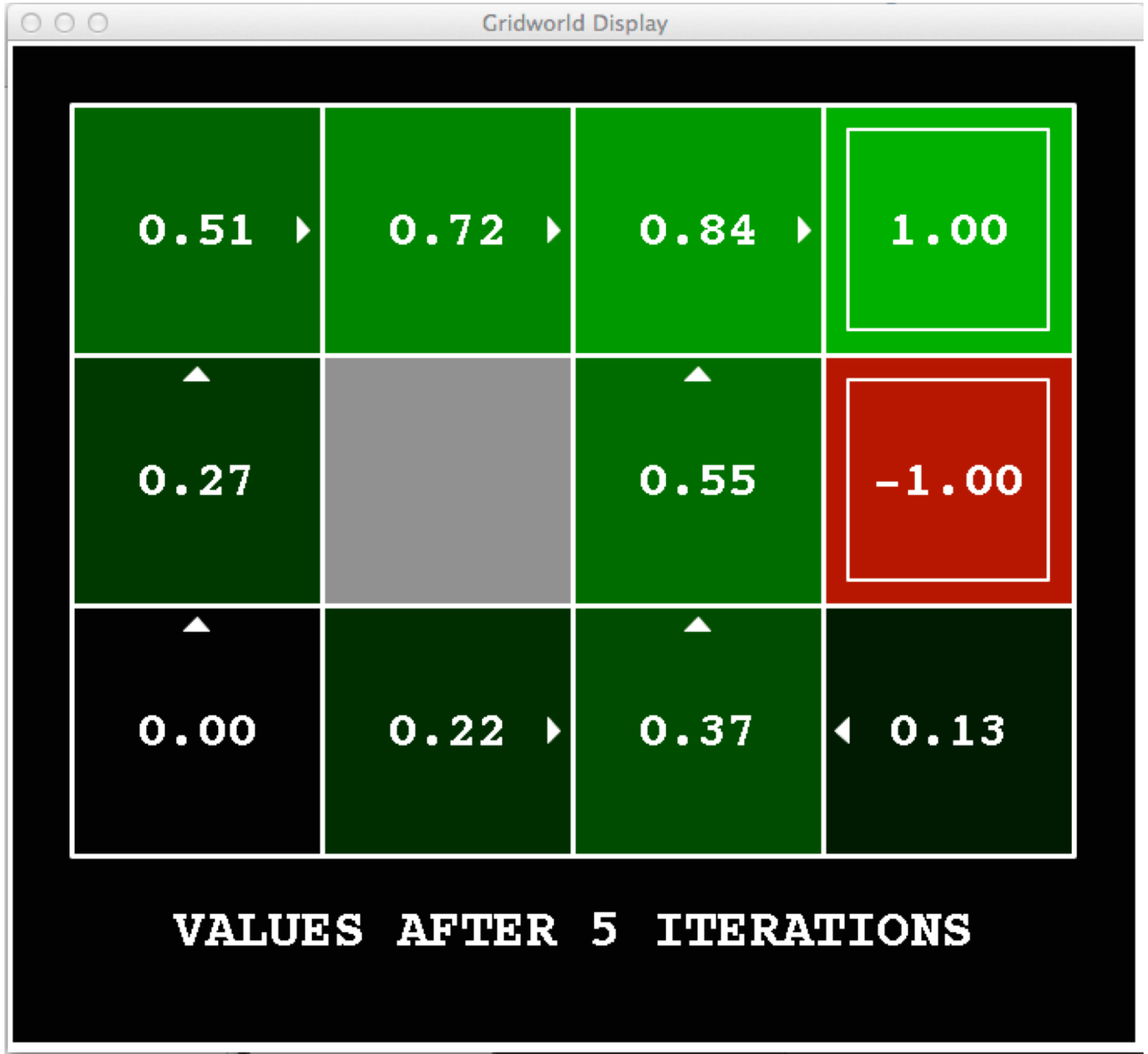
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



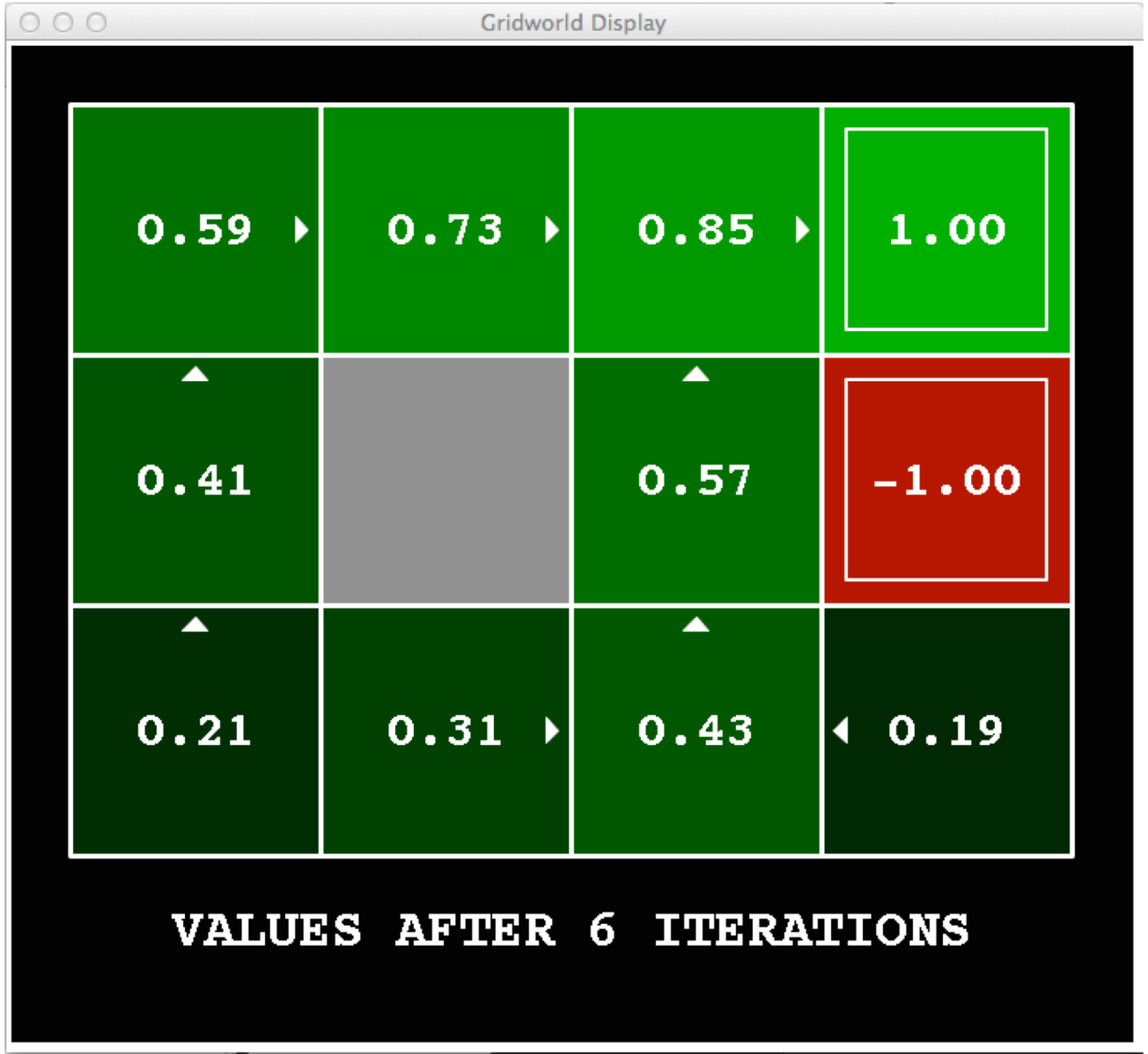
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



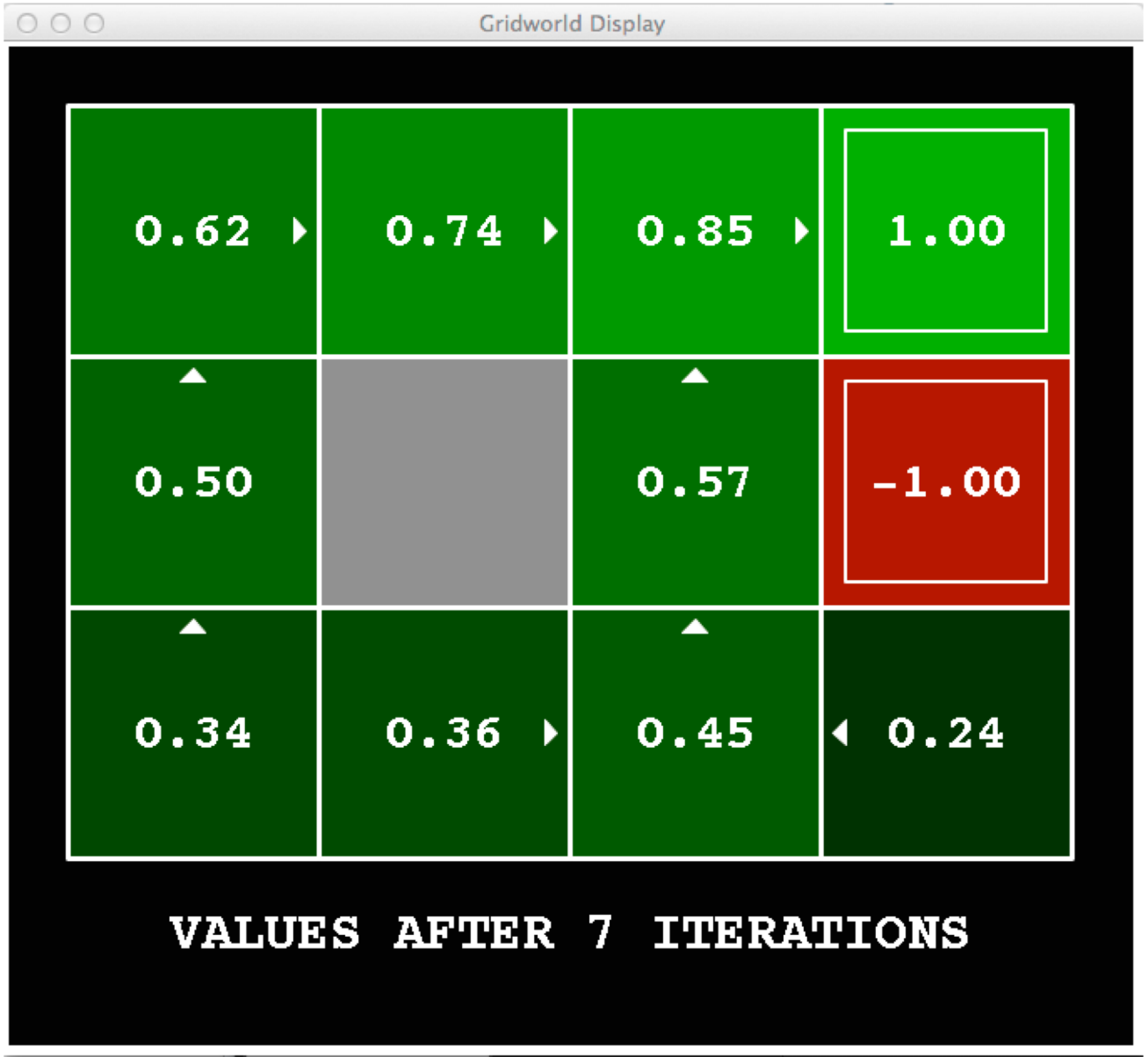
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



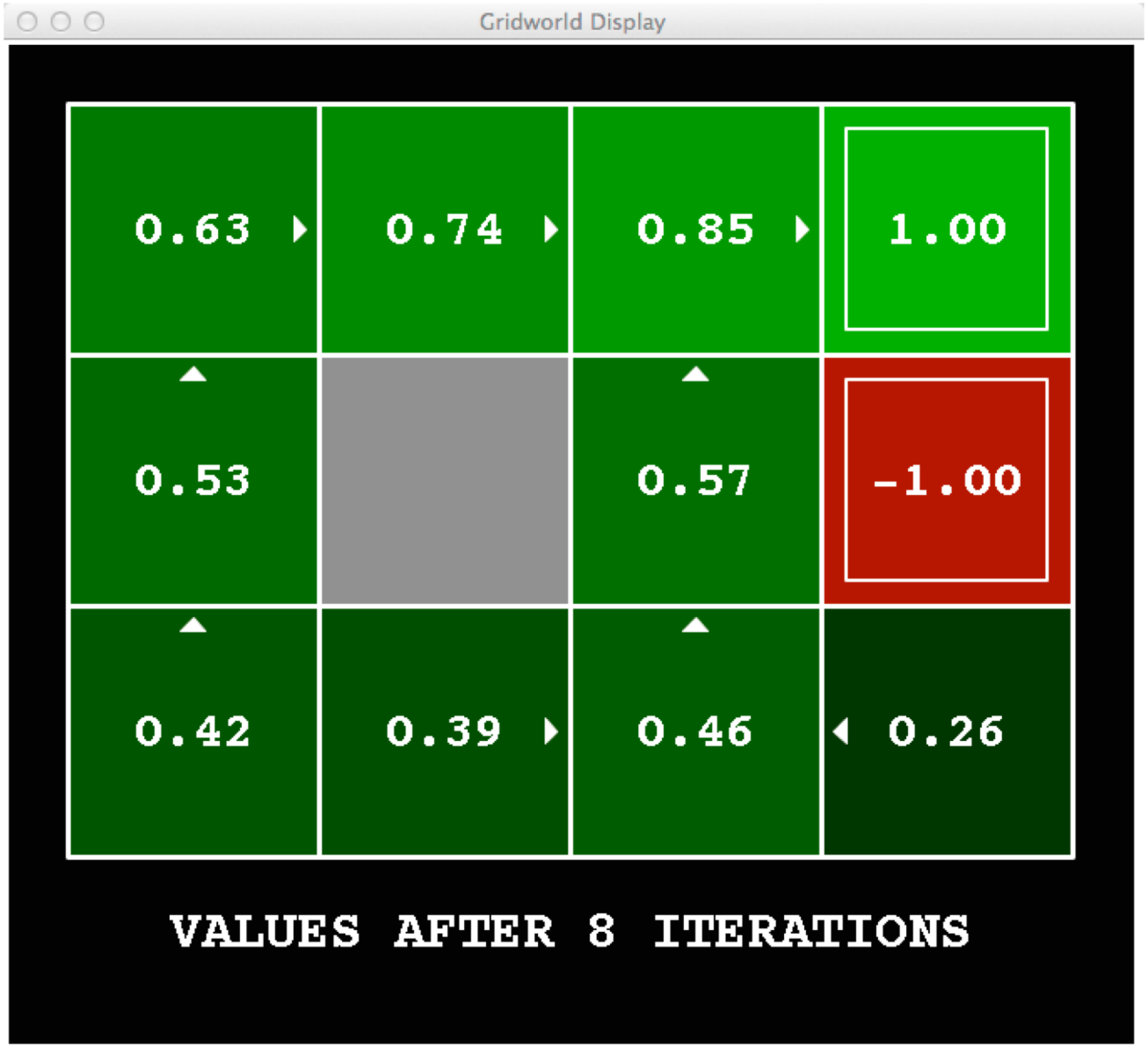
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



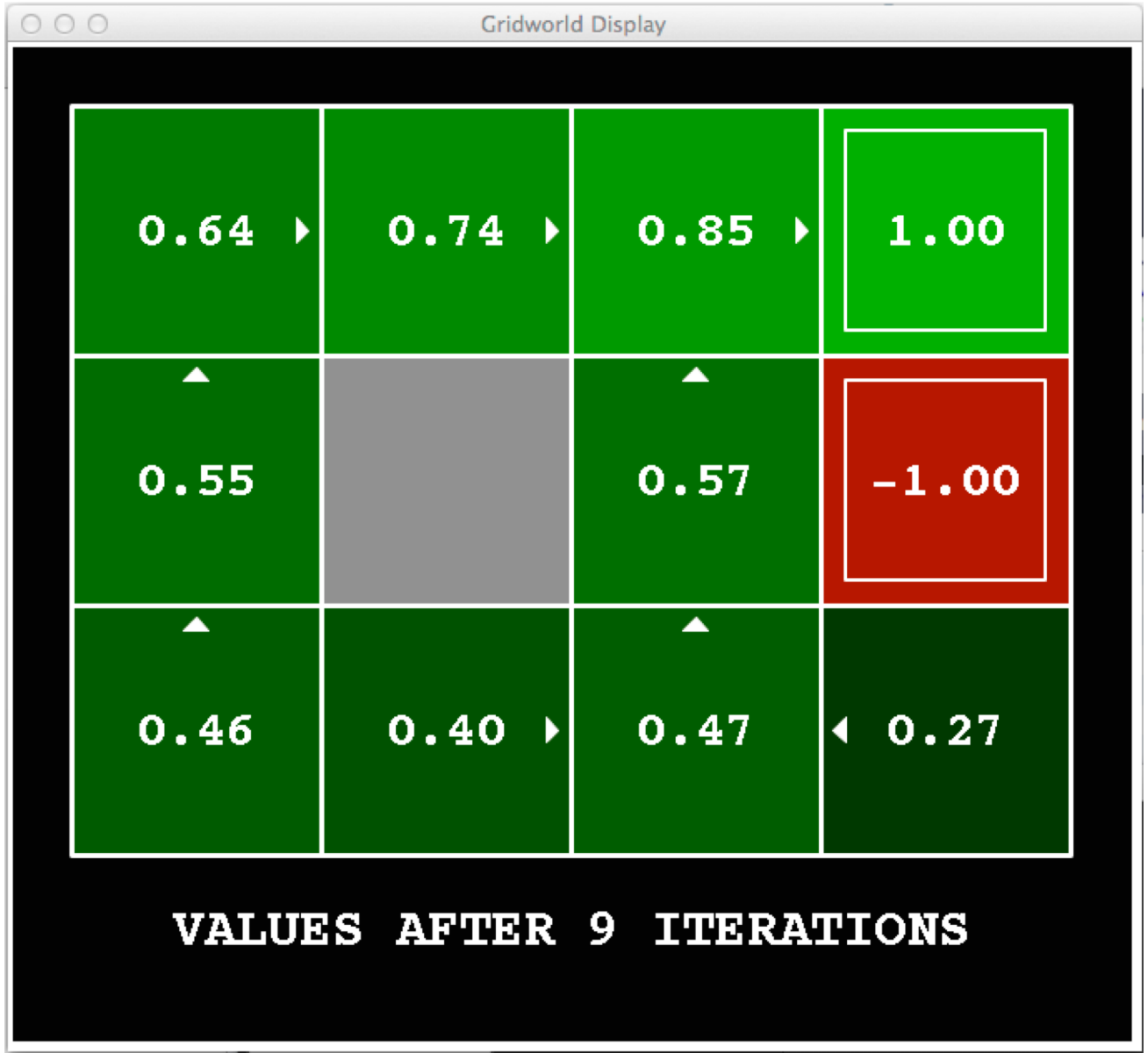
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



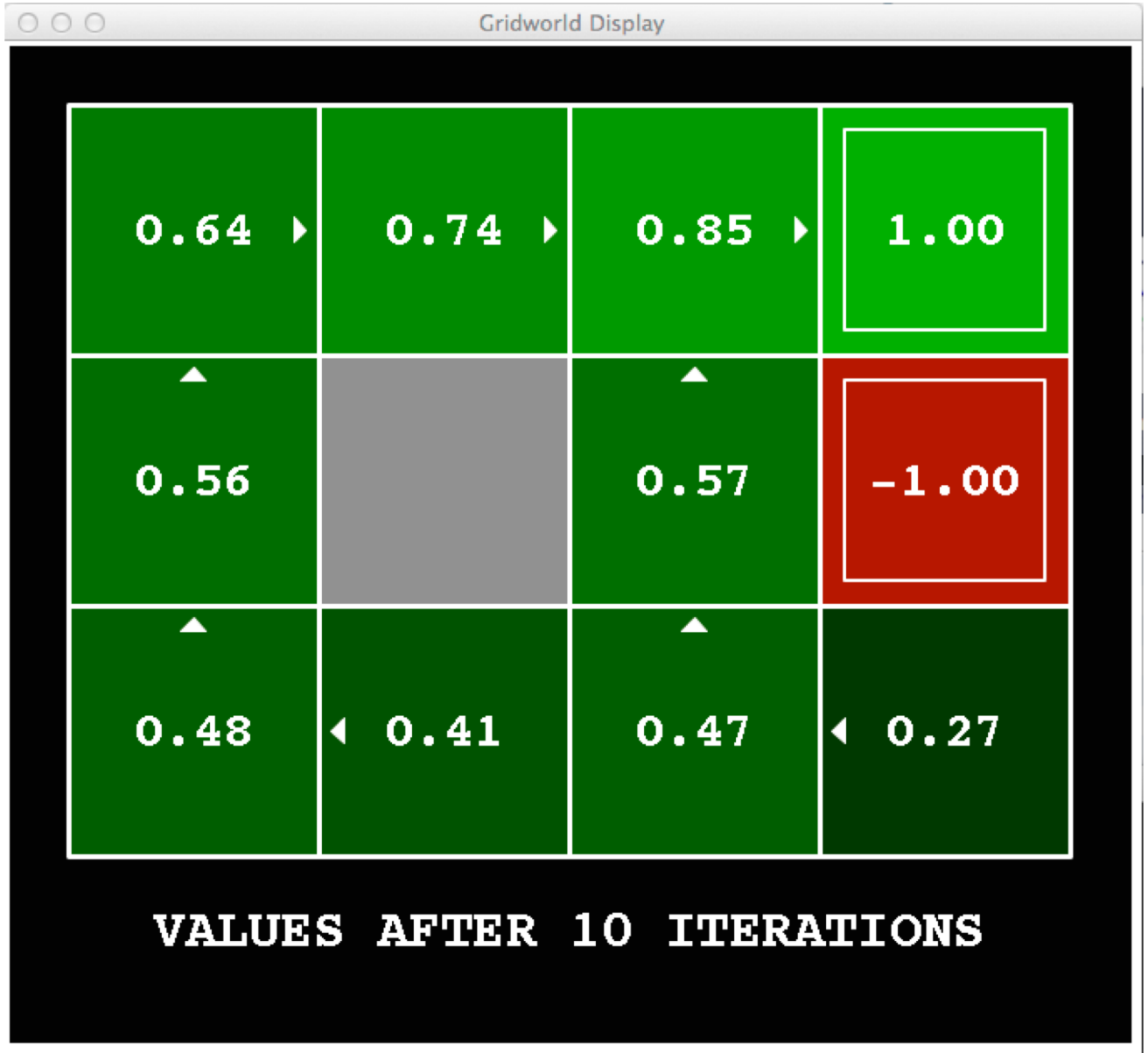
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



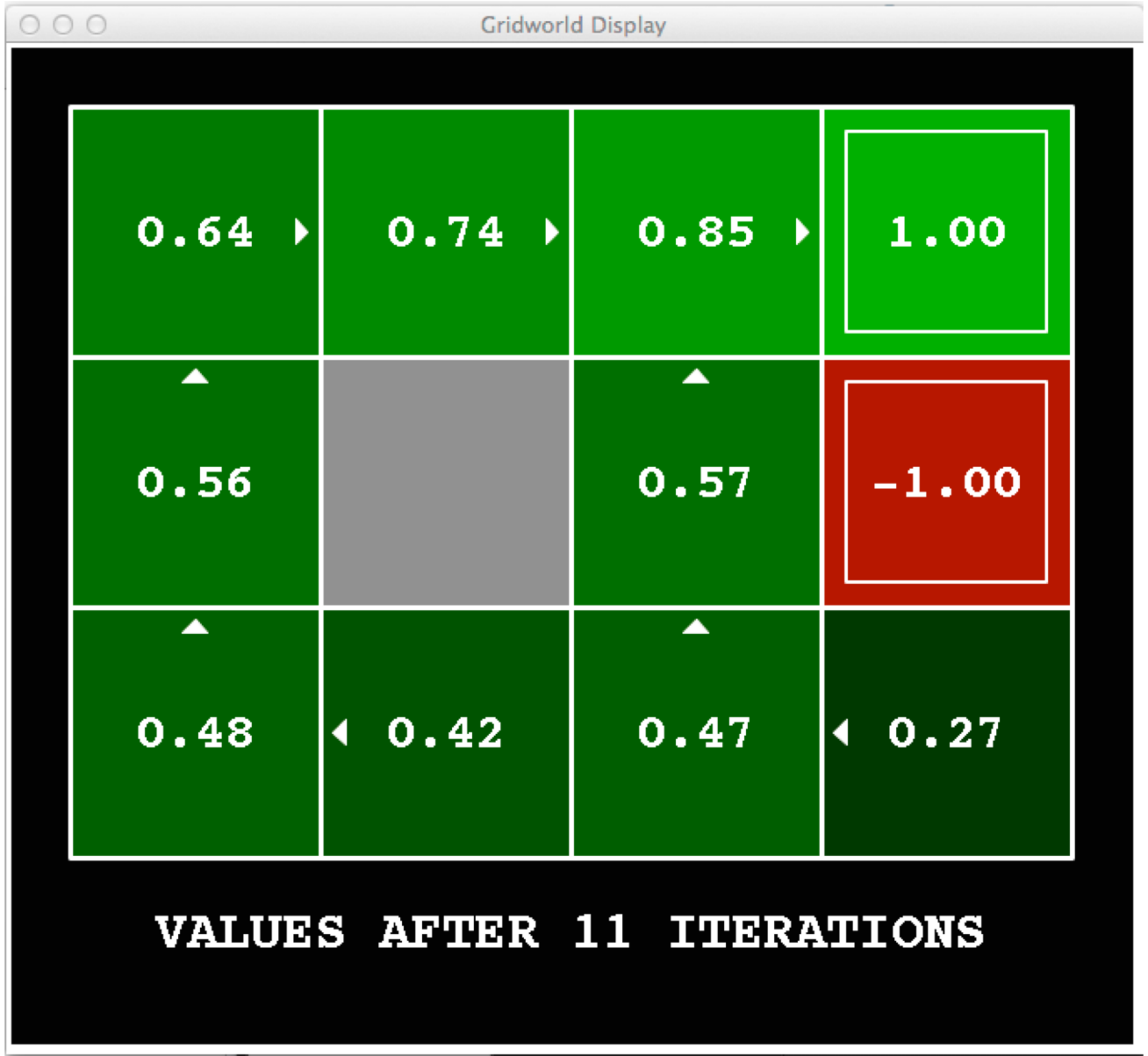
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



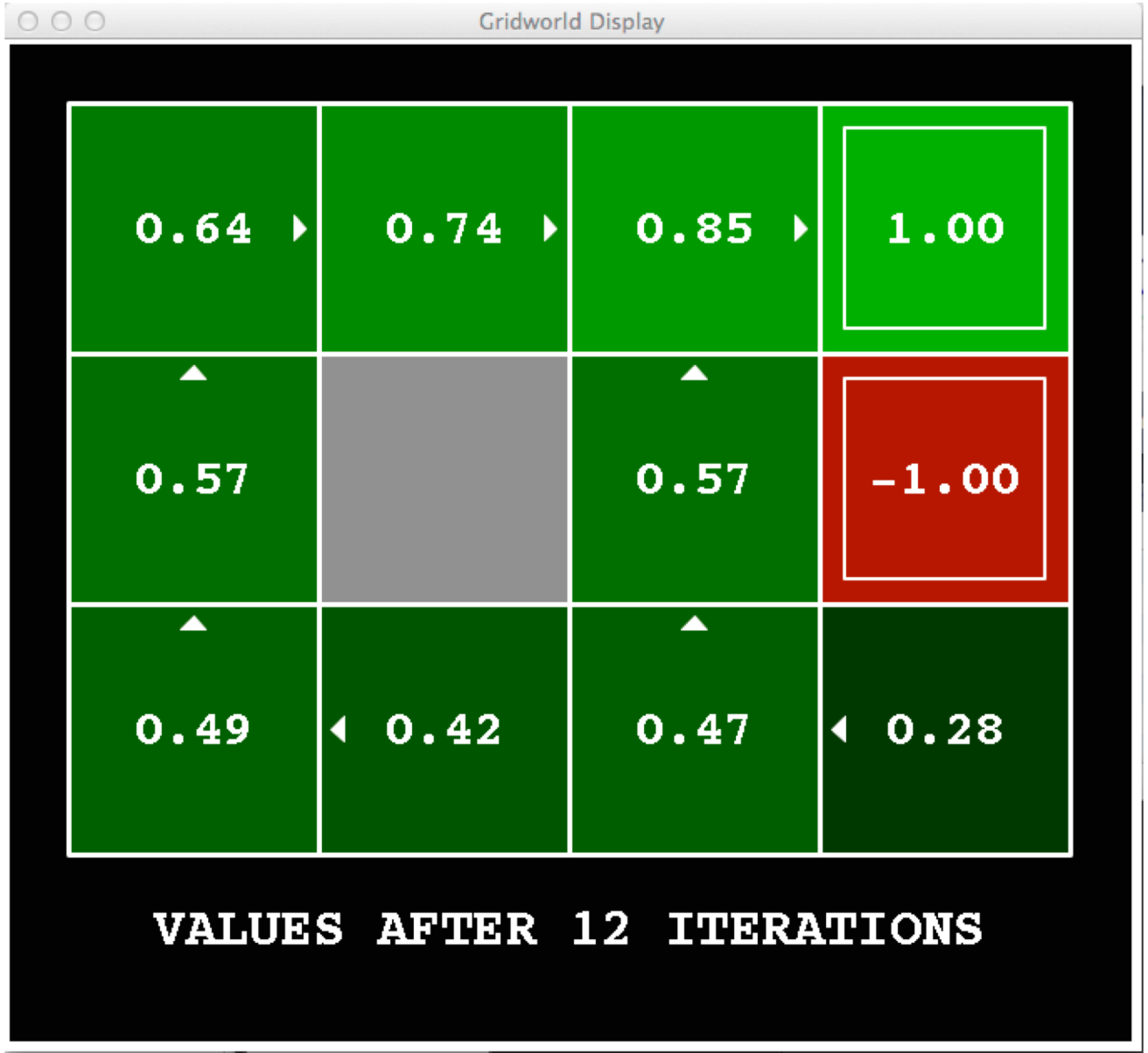
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



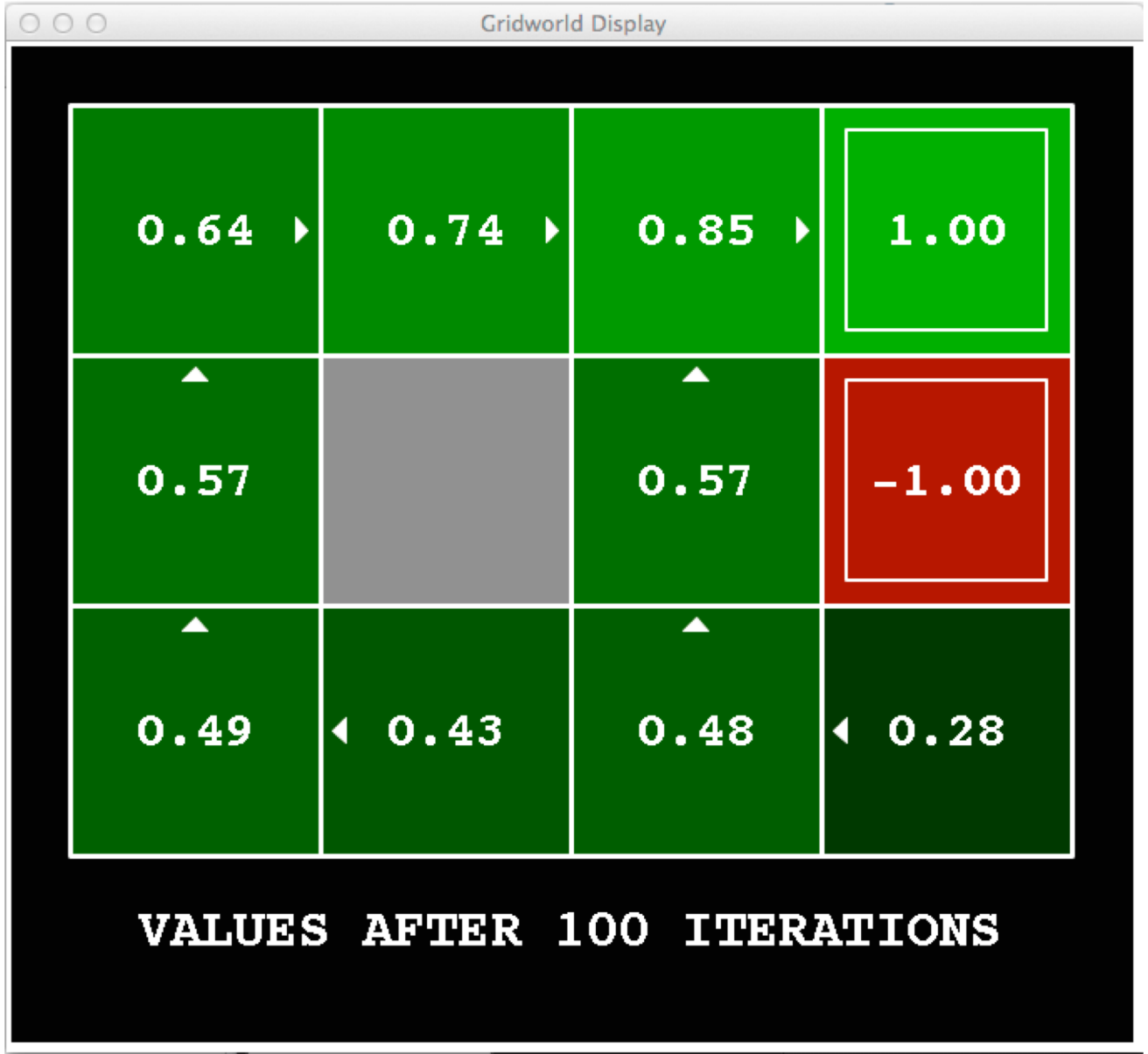
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



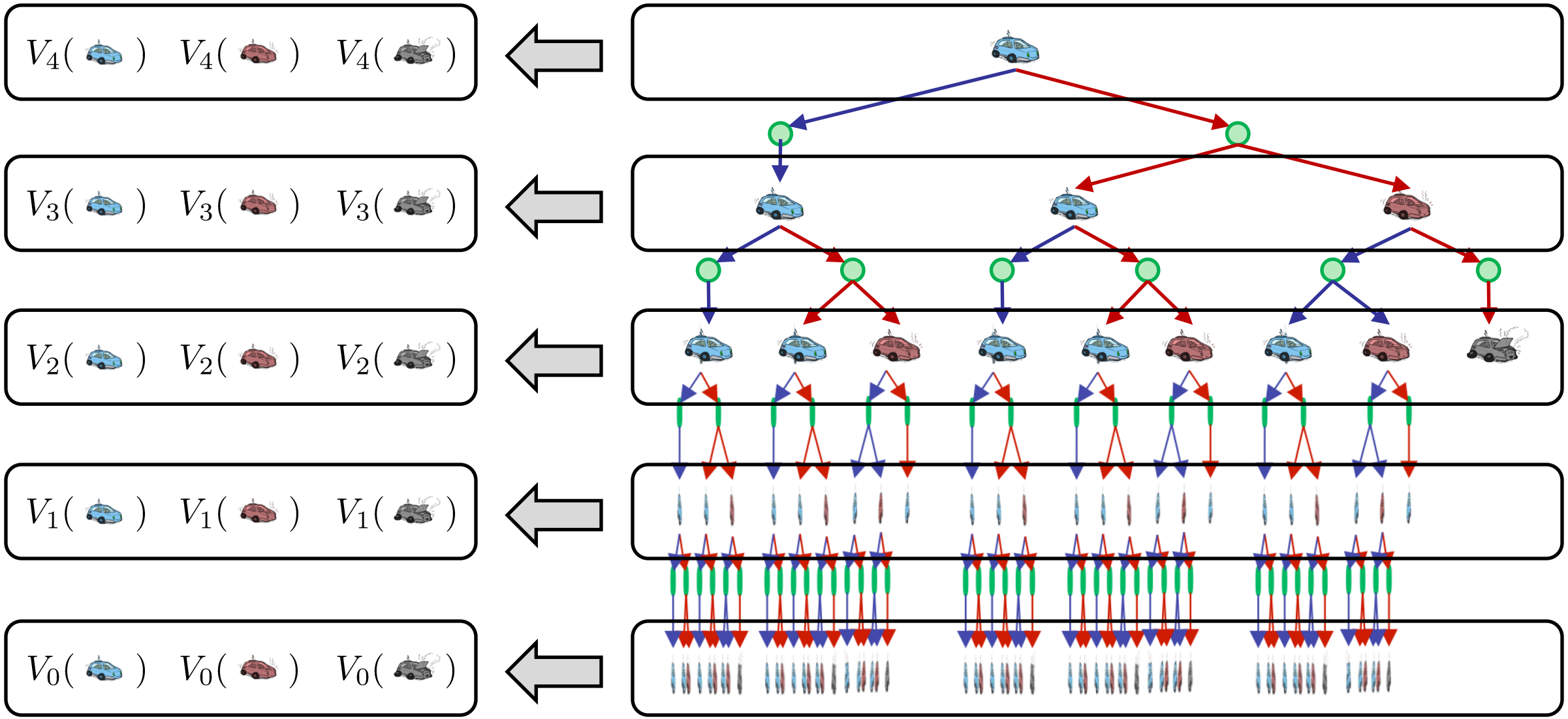
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Computing Time-Limited Values



Value Iteration

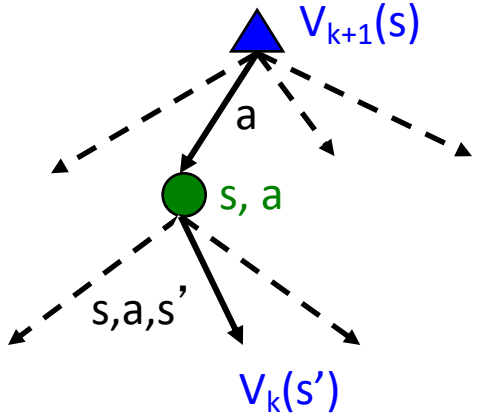
Value Iteration

1. For all states s, $V_0(s) = 0$
2. For each state s, given vector of V_k values, compute V_{k+1}

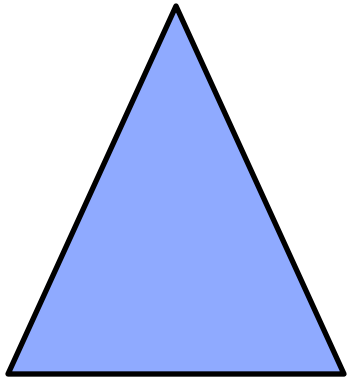
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

   - This is called Value Update or Bellman Update

3. Repeat until convergence



- Theorem: will converge to unique optimal values
  - Policy may converge long before values do



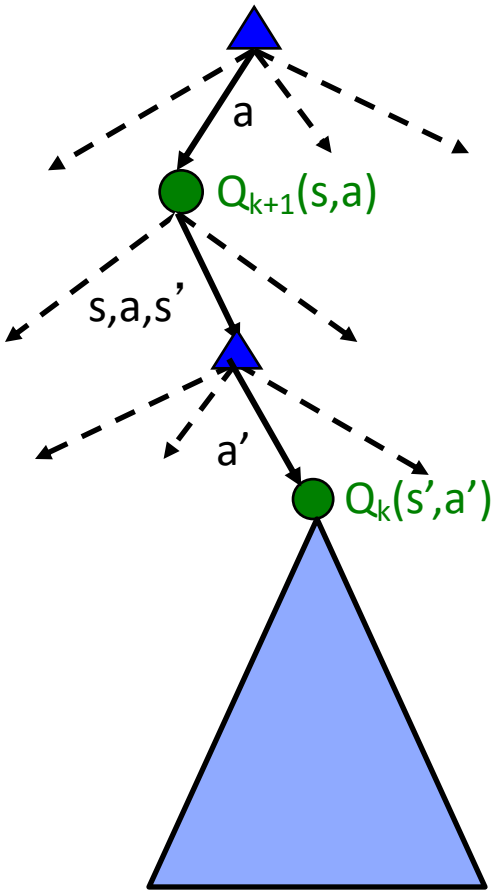
Value Iteration

1. For all q-states, $Q_0(s,a) = 0$
2. For all q-states, given vector of Q_k values, compute Q_{k+1}

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q_k(s',a')]$$

   - This is called Value Update or Bellman Update

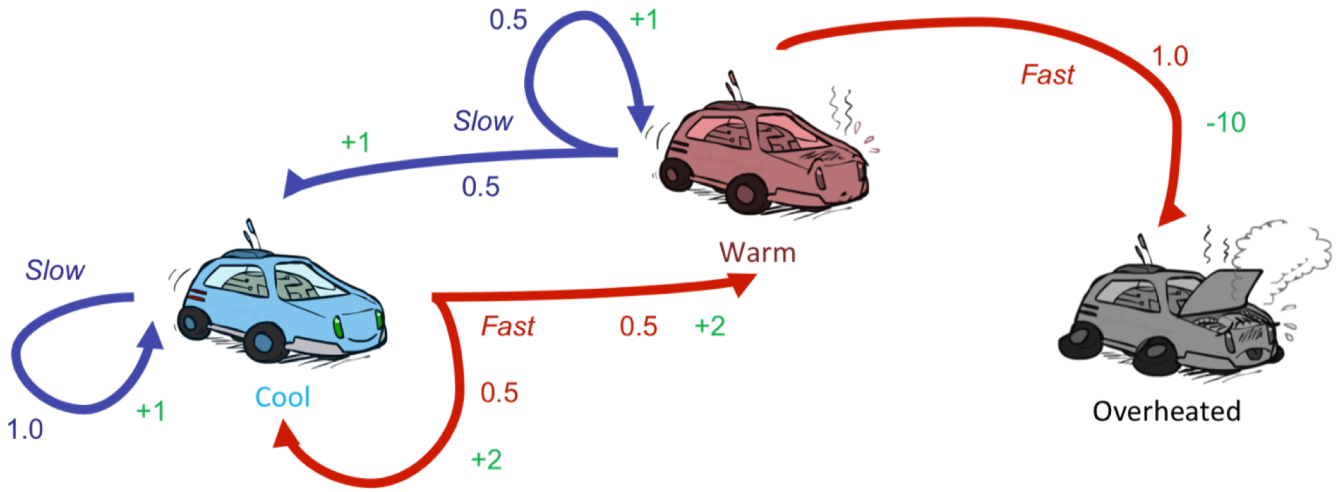
3. Repeat until convergence



- Theorem: will converge to unique optimal q-values
  - Policy may converge long before values do

Example: Value Iteration

V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0



Assume no discount!

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

Policy Extraction

Computing Actions from Values

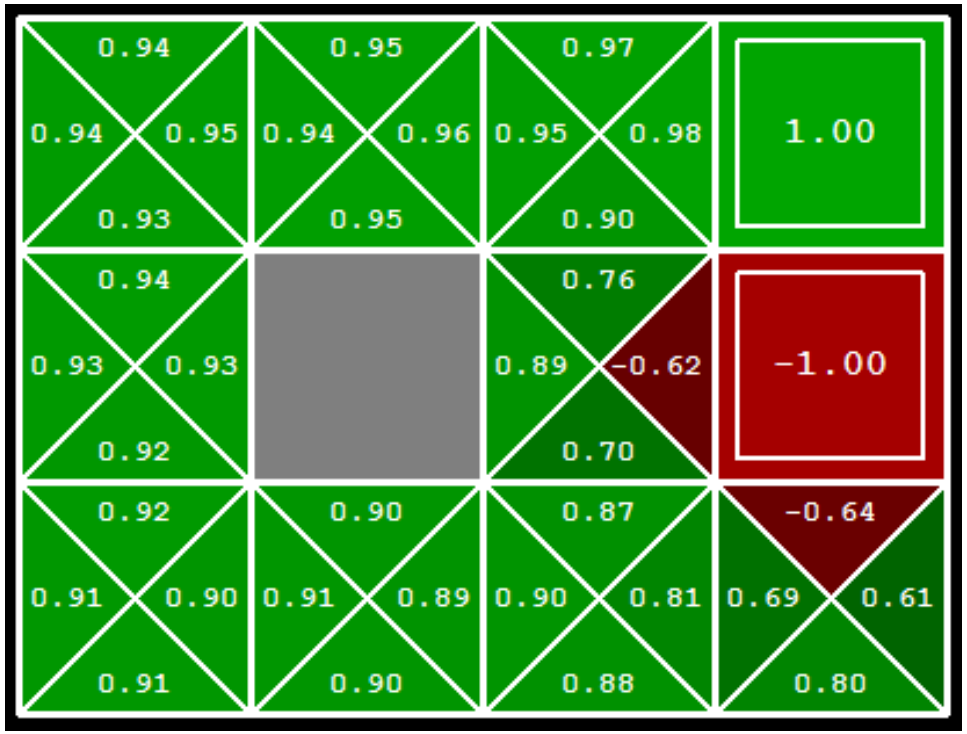
- Let's imagine we have the optimal values V*(s)
- How should we act?
  - It's not obvious!
- What action to take at each state?

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

- 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: Pseudocode

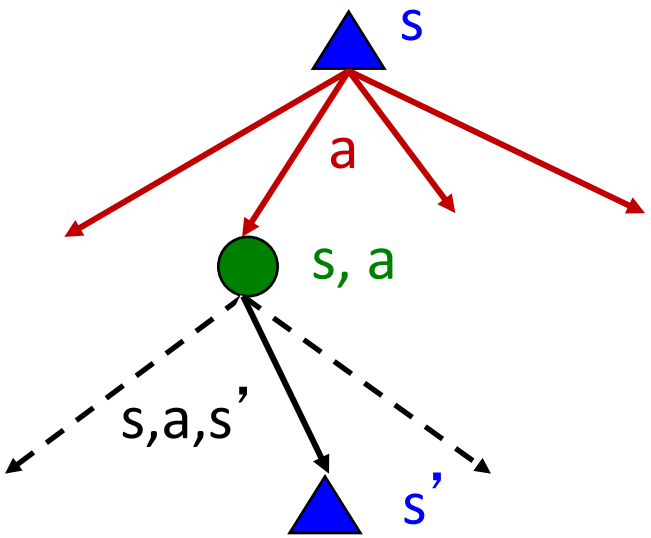
1: **Procedure** Value_Iteration(S,A,P,R,θ)
2: **Inputs**
3: S is the set of all states
4: A is the set of all actions
5: P is state transition function specifying $P(s'/s,a)$
6: R is a reward function $R(s,a,s')$
7: θ a threshold, $\theta>0$
8: **Output**
9: $\pi[S]$ approximately optimal policy
10: $V[S]$ value function
11: **Local**
12: real array $V_k[S]$ is a sequence of value functions
13: action array $\pi[S]$
14: assign $V_0[S]$ arbitrarily
15: $k \leftarrow 0$
16: **repeat**
17: $k \leftarrow k+1$
18: **for each** state s **do**
19: $V_k[s] = max_a \sum_{s'} P(s'/s,a) (R(s,a,s')+ \gamma V_{k-1}[s'])$
20: **until** $\forall s \ |V_k[s]-V_{k-1}[s]| < \theta$
21: **for each** state s **do**
22: $\pi[s] = argmax_a \sum_{s'} P(s'/s,a) (R(s,a,s')+ \gamma V_k[s'])$
23: **return** π, V_k

Value Iteration: Complexity

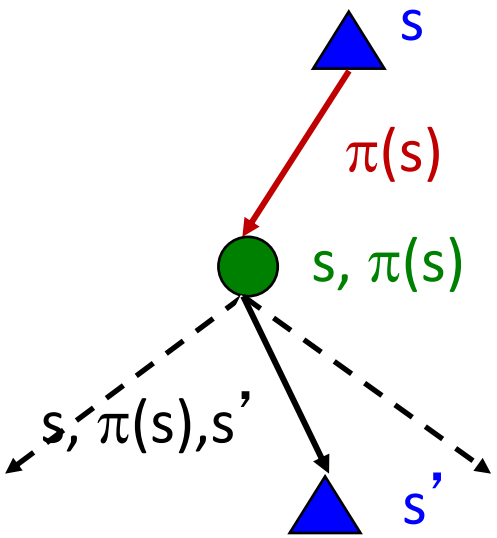
- Problem size:
  - $|A|$ actions and $|S|$ states
- Each Iteration
  - Computation: $O(|A| \cdot |S|^2)$
- Space: $O(|S|)$
- Num of iterations
  - Can be exponential in the discount factor γ

Fixed Policies

Do the optimal action



Do what π says to do



- 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

Policy Evaluation

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

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

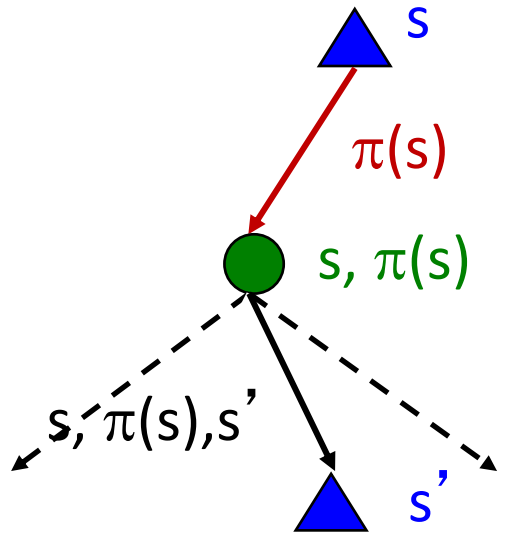
- How do we calculate the V's for a fixed policy π?

  1. For all states s, $V_0^{\pi}(s) = 0$

  2. For each state s, given vector of V_k values, compute V_{k+1}

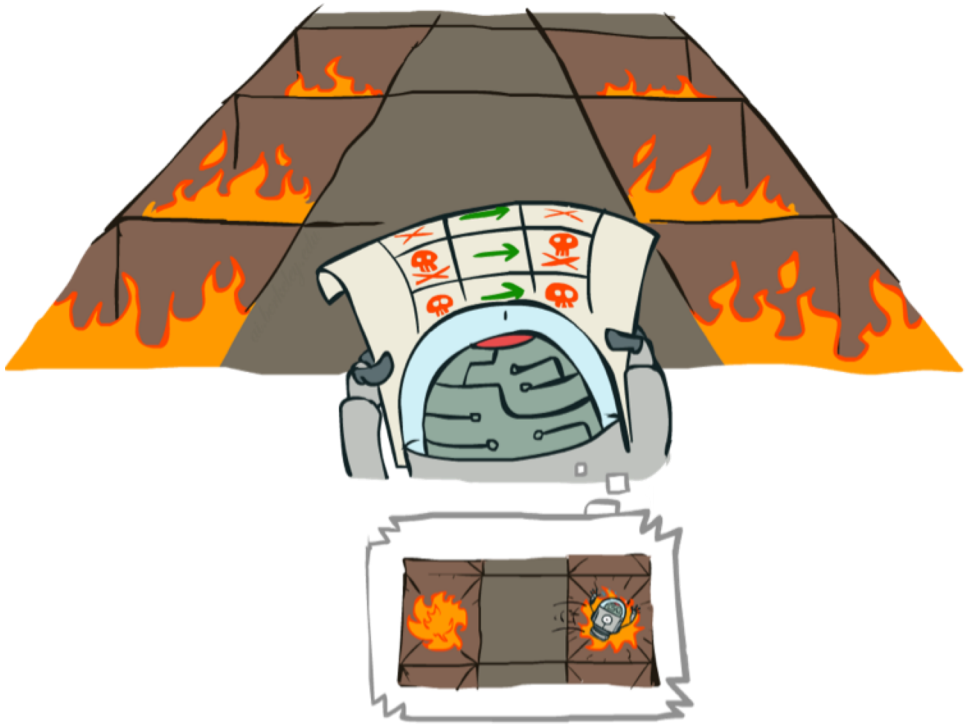
$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

  3. Repeat until convergence

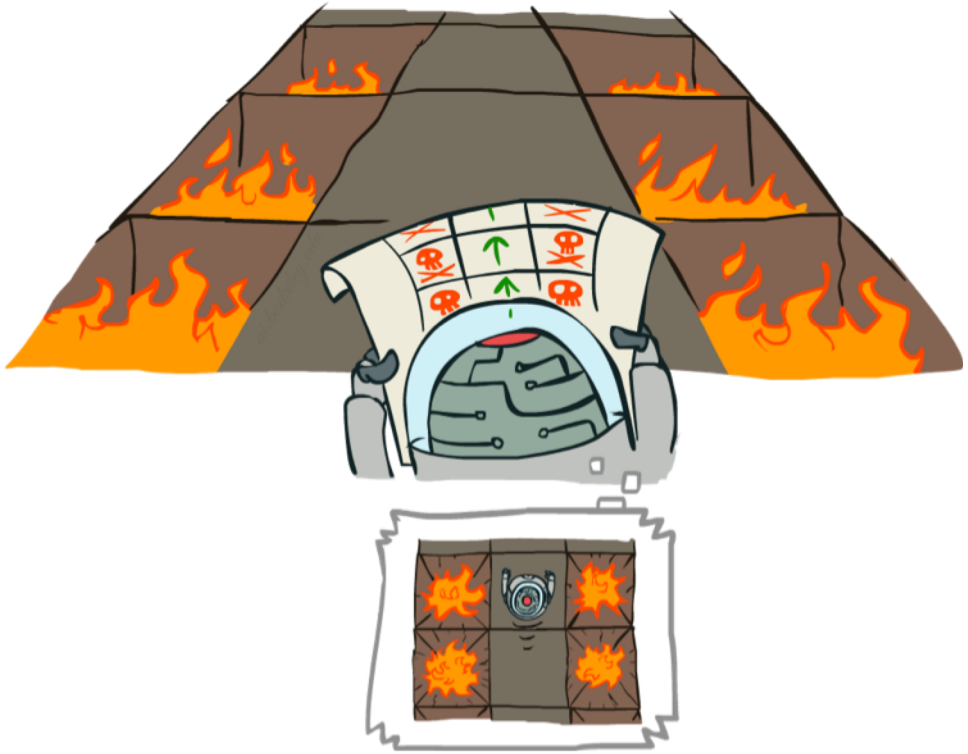


Example: Policy Evaluation

Always Go Right



Always Go Forward



Example: Policy Evaluation

Always Go Right

Always Go Forward

Summary

- 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

Summary

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

Reading

- Read Sections 17.1, 17.2 in the AIMA textbook
- Optional Reading: 17.3