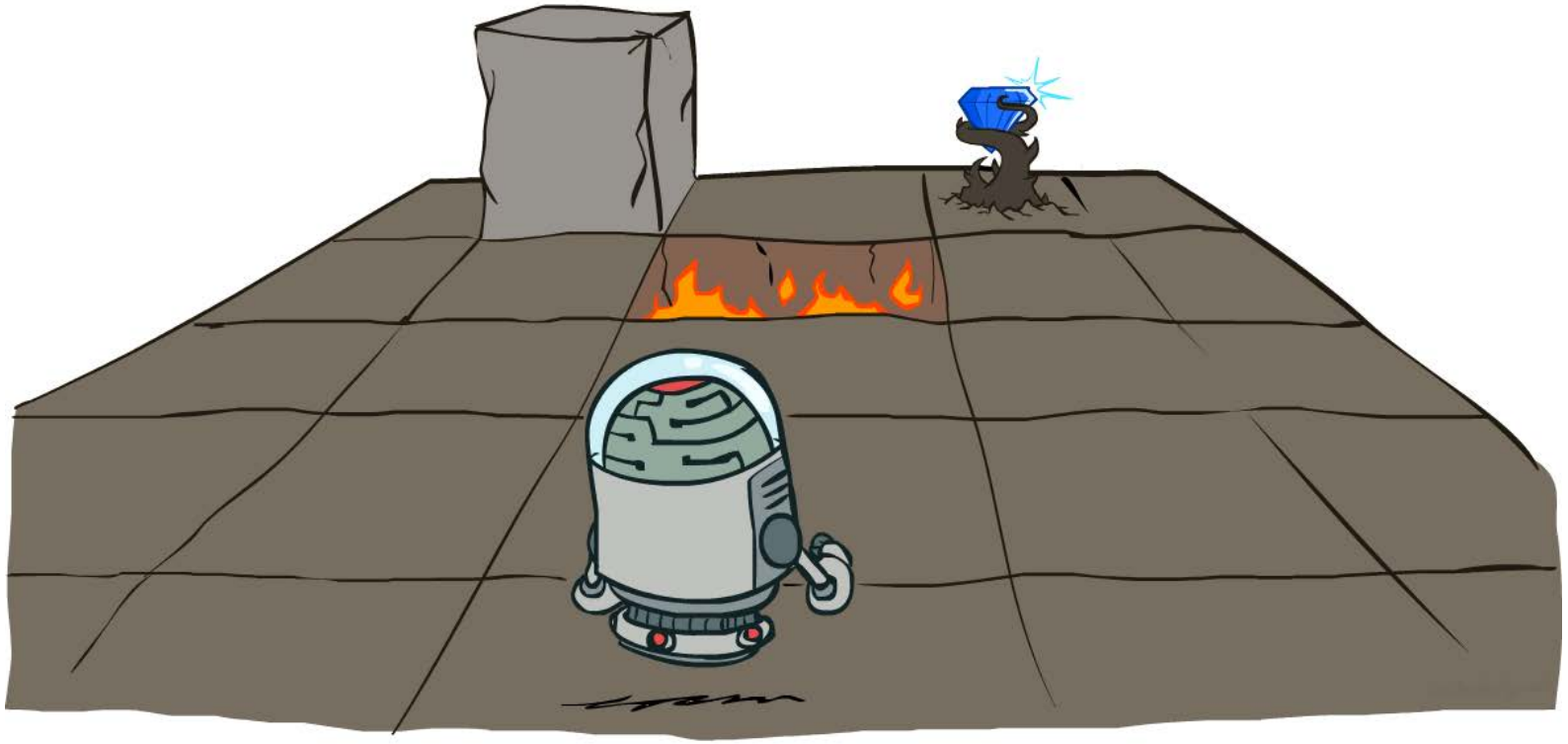
MDPs are generalizations of search problems where you're not entirely sure what you're actions are going to do

We need non-deterministic search because we know which actions are available, but we aren't exactly sure what outcomes will occur. You also might know the possible outcomes, but you don't know which outcome you'll get.

CS 188: Artificial Intelligence

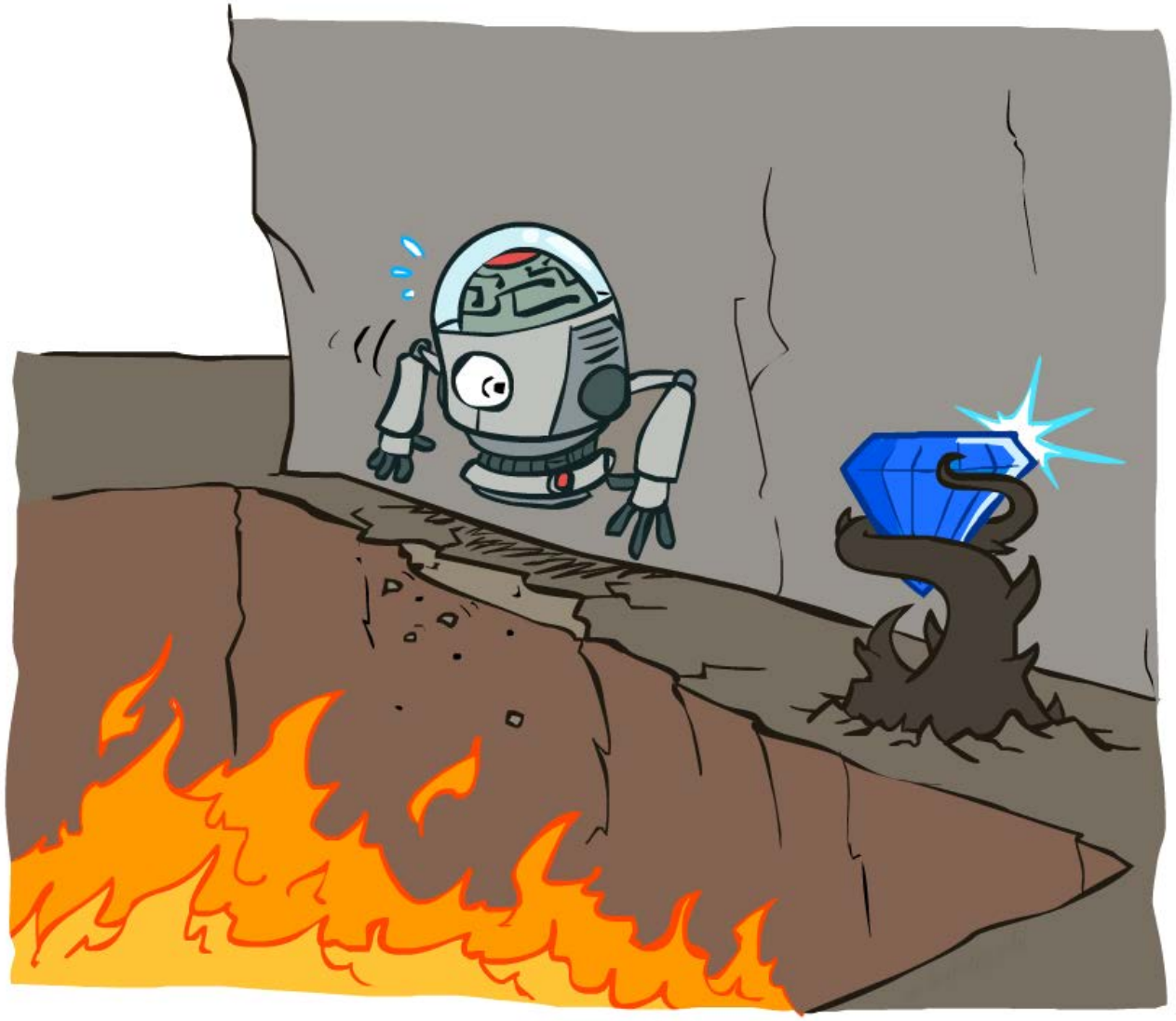
Markov Decision Processes



Instructors: Dan Klein and Pieter Abbeel

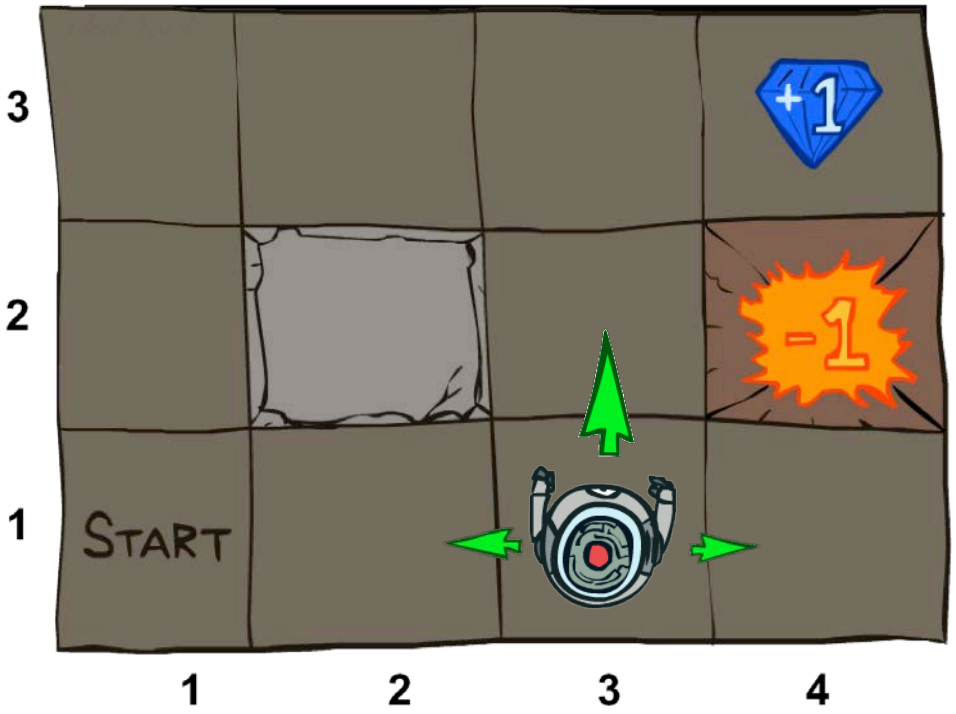
University of California, Berkeley

Non-Deterministic Search



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

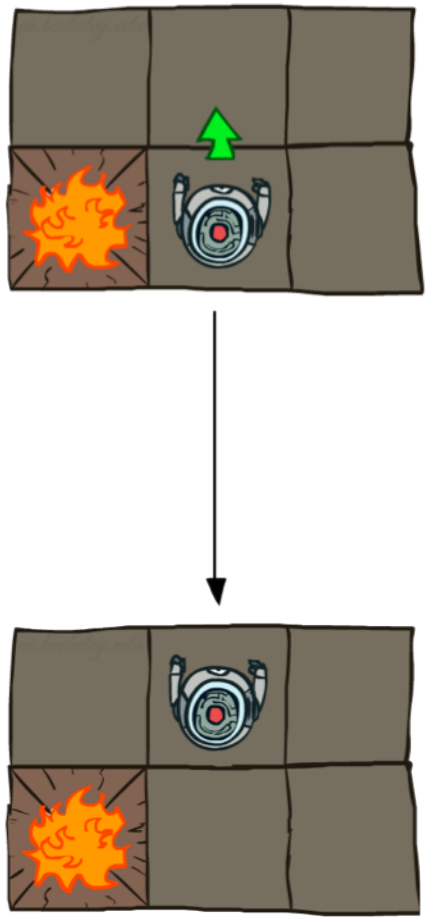


In a deterministic we could have solved this problem with search, but in MDP it's non-deterministic

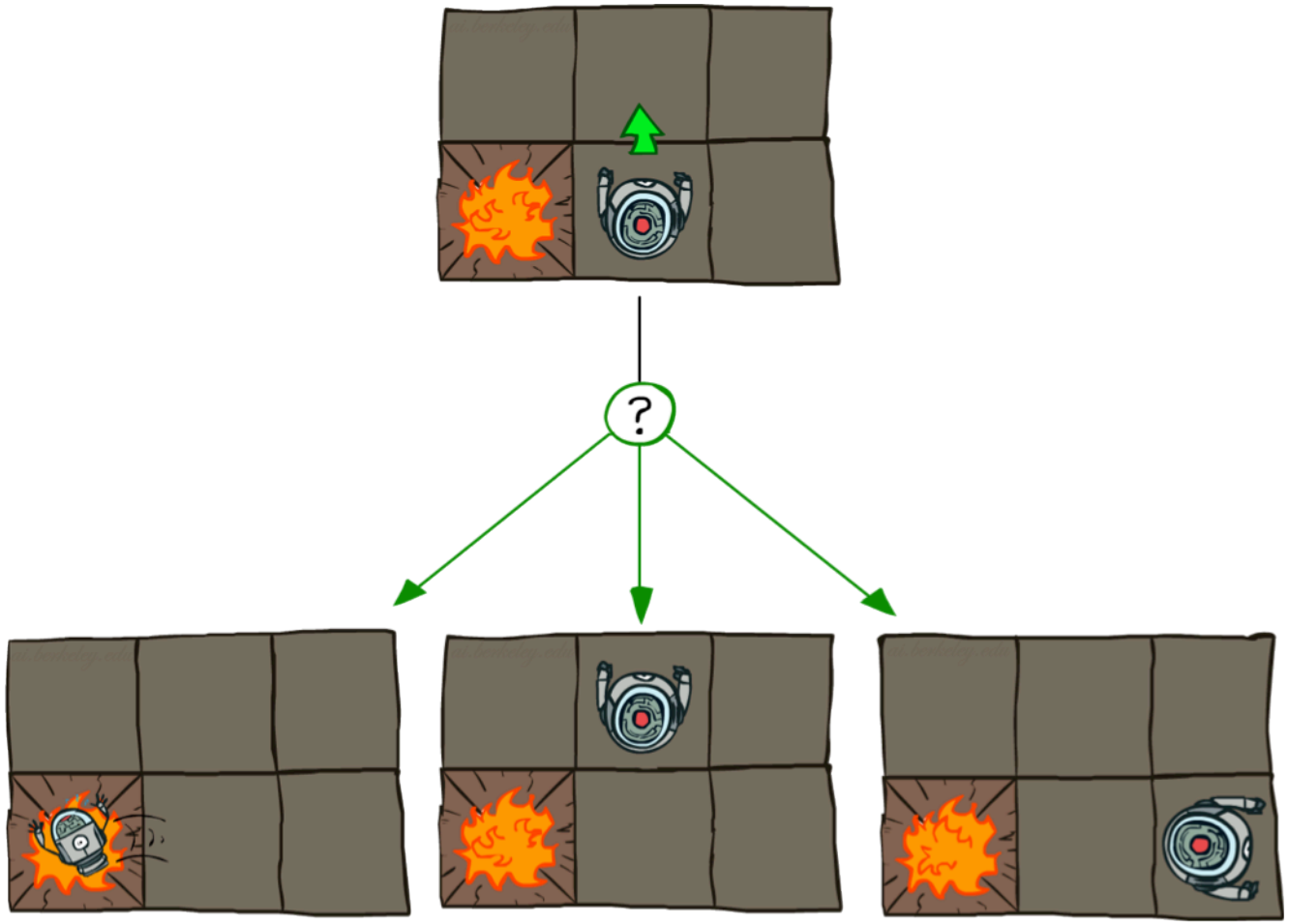
In MDP we KNOW the possible set of outcomes and we know the PROBABILITY at which they will occur!

Grid World Actions

Deterministic Grid World



Stochastic Grid World

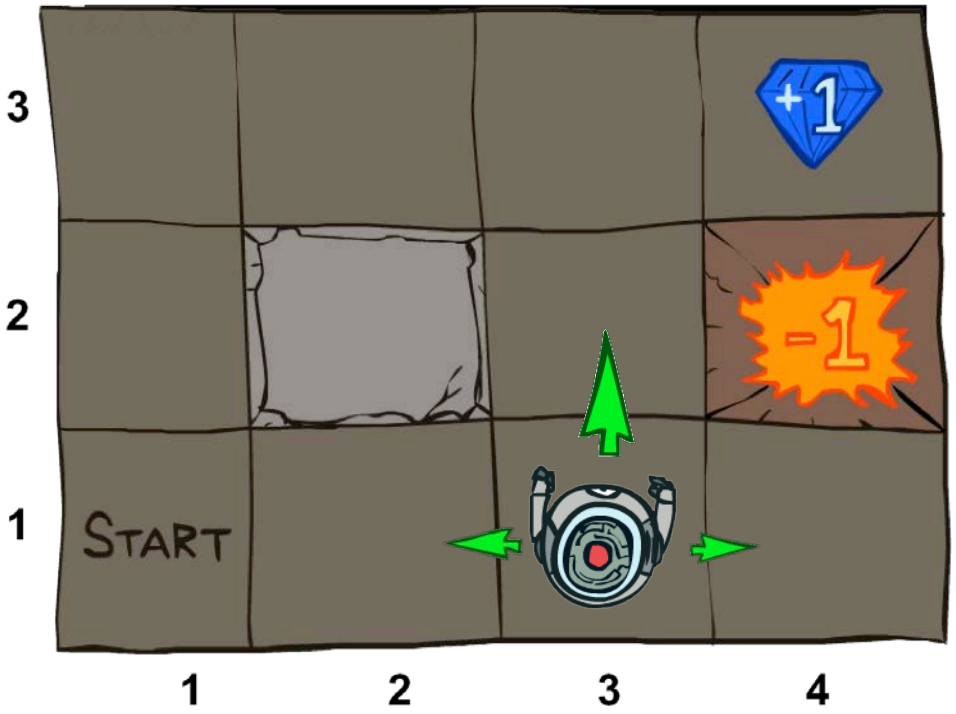


Markov Decision Processes

- 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 $R(s, a, s')$
    - Sometimes just R(s) or R(s')
  - A start state
  - Maybe a terminal state

We know this ahead of time...Well the agent doesn't know it, but these are preset

Need not have a terminal state. Might go on "forever"



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

Can't solve this with A*

[Demo – gridworld manual intro (L8D1)]

Video of Demo Gridworld Manual Intro

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 action outcomes depend only on the current state

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

$$=$$

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

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

The future state depends on our current state and the action we take. That's it. It doesn't depend on the previous state

Andrey Markov
(1856-1922)

Another difference from search: In search, the agent looks at the search problem and does a bunch of off-line computation to come up with a plan (i.e. sequence of actions). Everything would happen to plan. MDP is noisier and you can't have a plan! In MDPs, we don't have plans, but we have policies (which are similar). The policy takes a state as input and the output is an action.

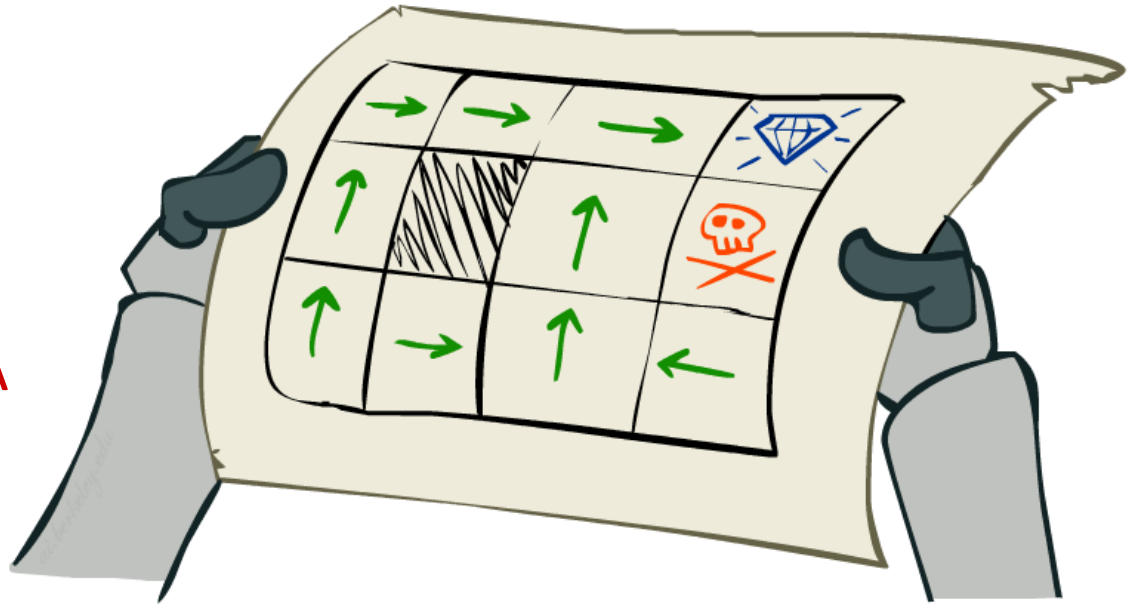
Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal

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

the computation was done in creating the policy. Here, we just execute the policy based on the pre-calculated policy.

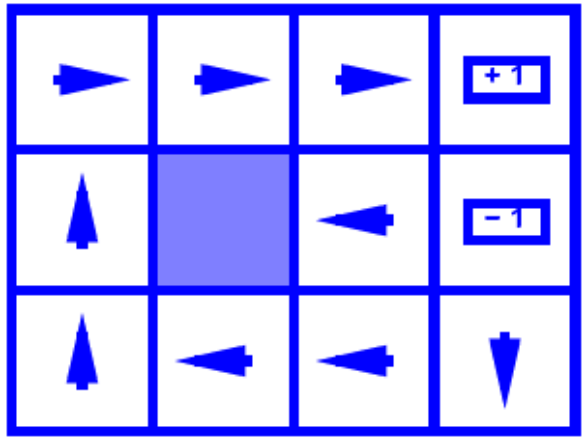
- Expectimax didn't compute entire policies
  - It computed the action for a single state only



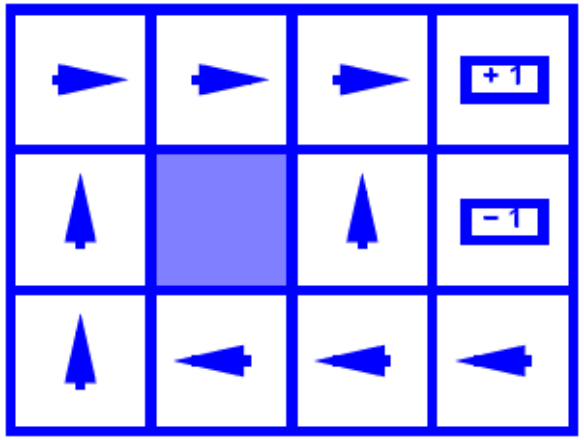
Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

Optimal Policies

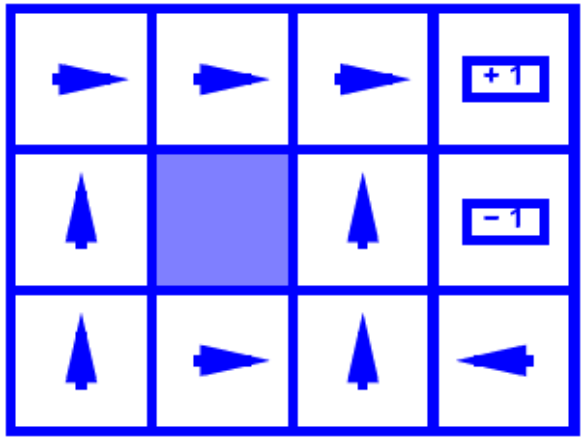
The reward for doing any step is so low that's it's not worth risking going into the pit. We have patience...



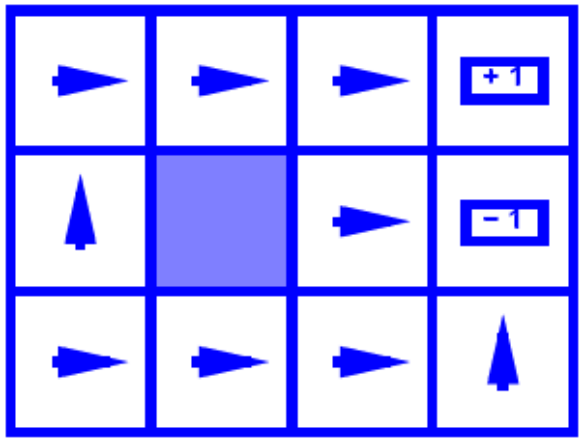
R(s) = -0.01



R(s) = -0.03



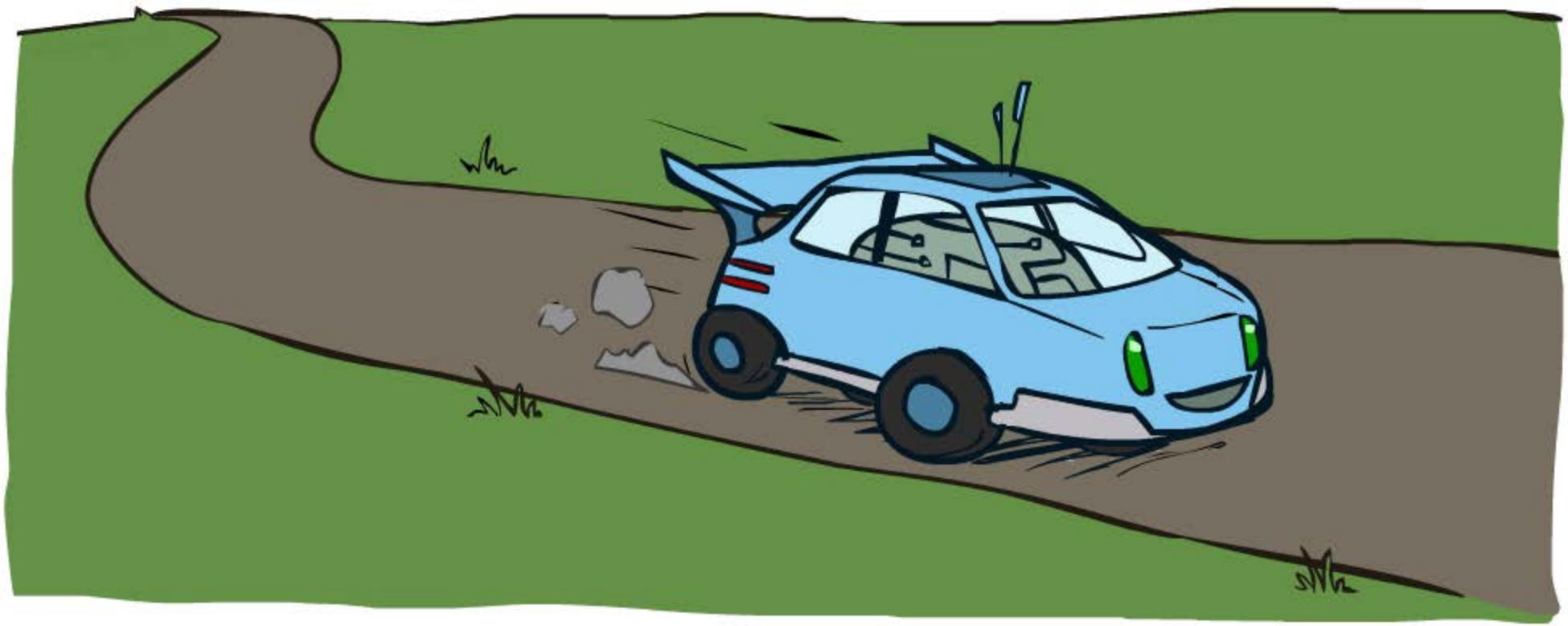
R(s) = -0.4



R(s) = -2.0

The reward for doing any step is so high, it's best to try to end the game, even if it means going into the pit!

Example: Racing

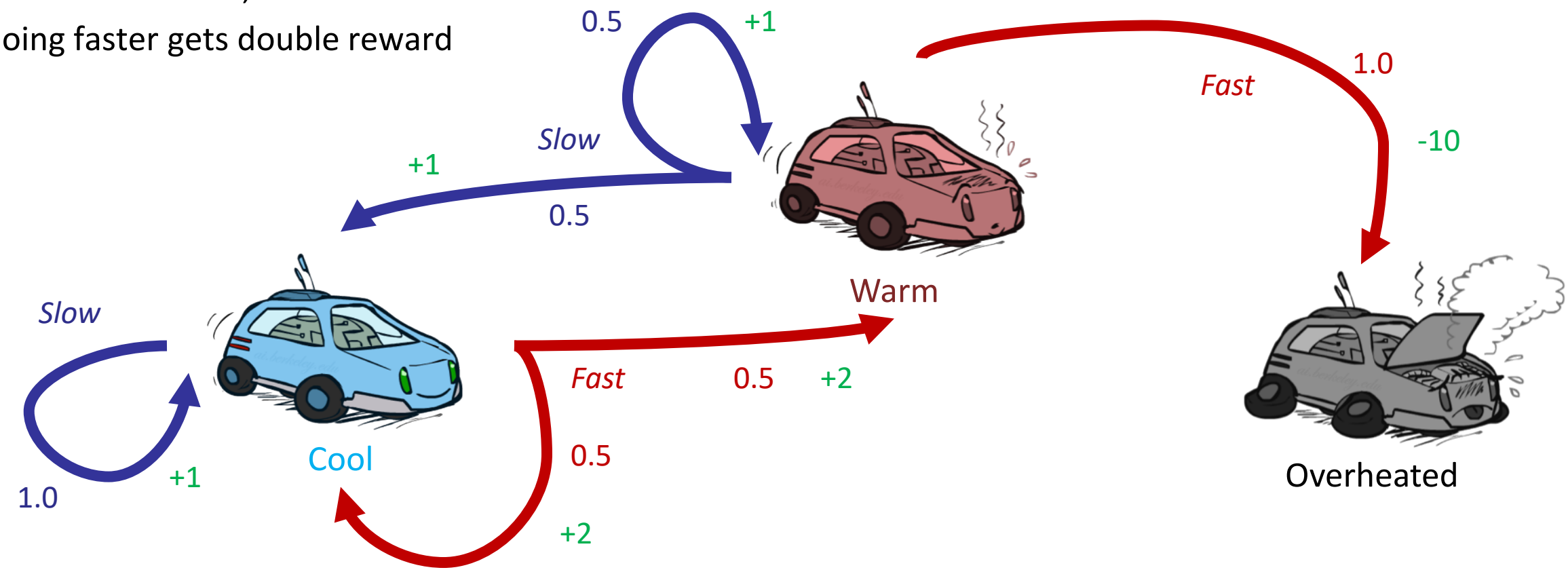


Example: Racing

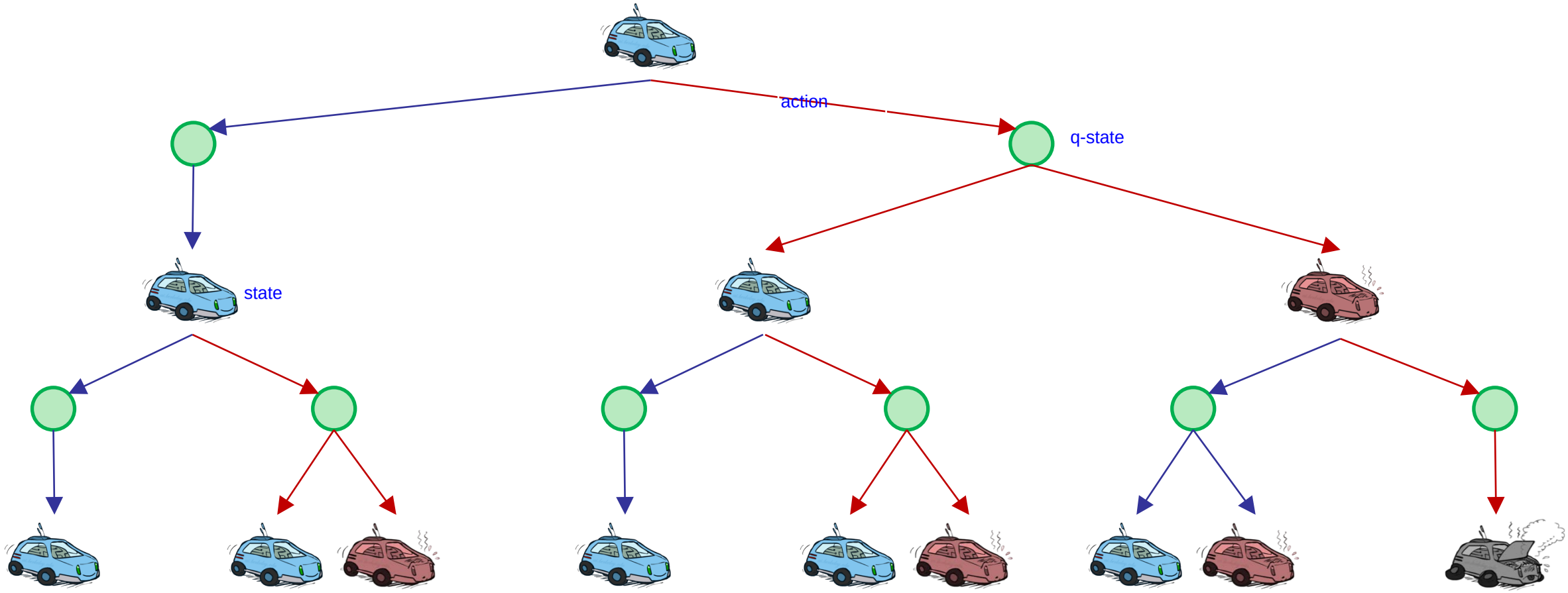
- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward

The rewards happen on the transitions. It doesn't matter if you're cool, warm or overheated.

If you're cool, you get +1 for going slow.
If you're cool, you get +2 for going fast.
If you're warm and you go fast, you get -10

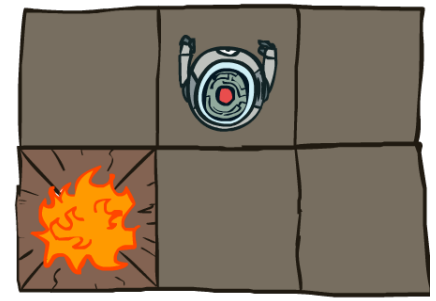


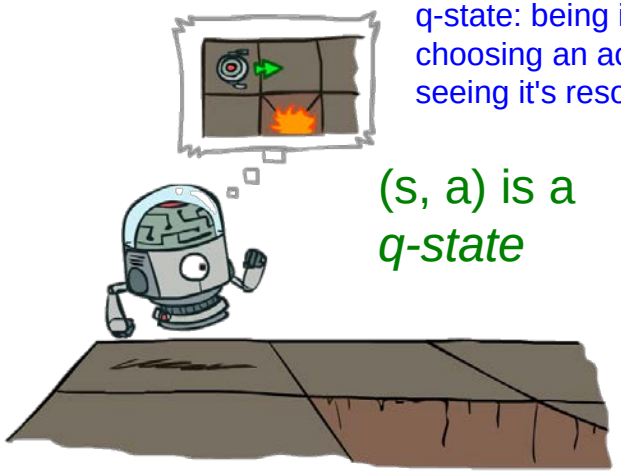
Racing Search Tree



MDP Search Trees

- Each MDP state projects an expectimax-like search tree





q-state: being in a state, choosing an action, but not yet seeing it's resolution/transition

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

s

s is a *state*

a

s, a

multiple outcomes could happen

s,a,s'

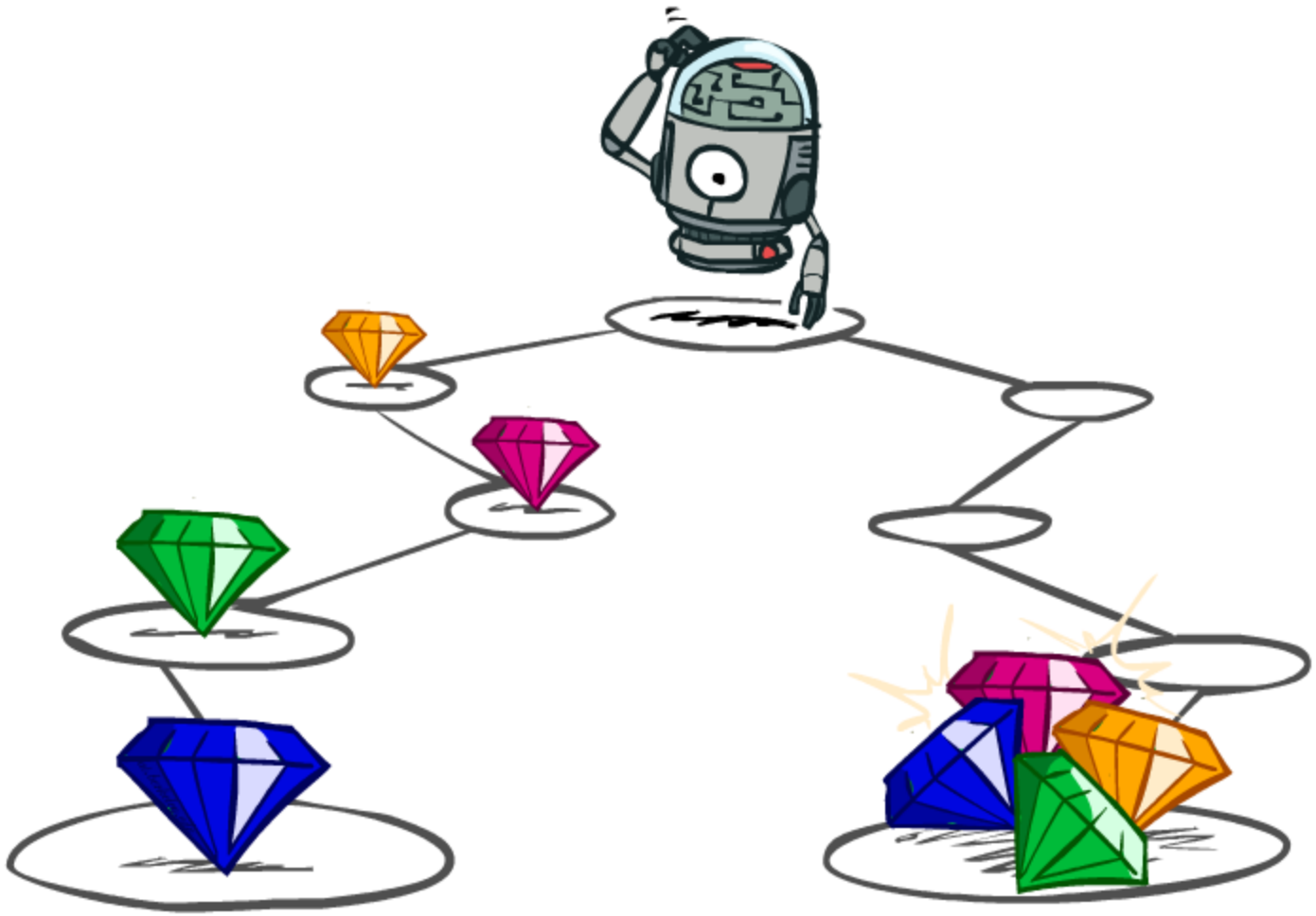
s'

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

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

R(s,a,s')

Utilities of Sequences



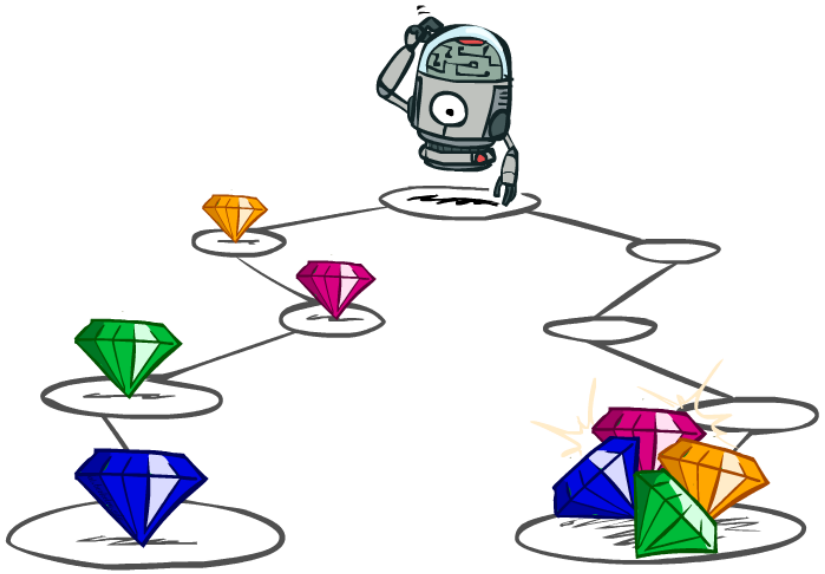
Utilities of Sequences

- What preferences should an agent have over reward sequences?

  There's a tradeoff between wanting rewards now vs. later

- 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

1

Worth Now

γ

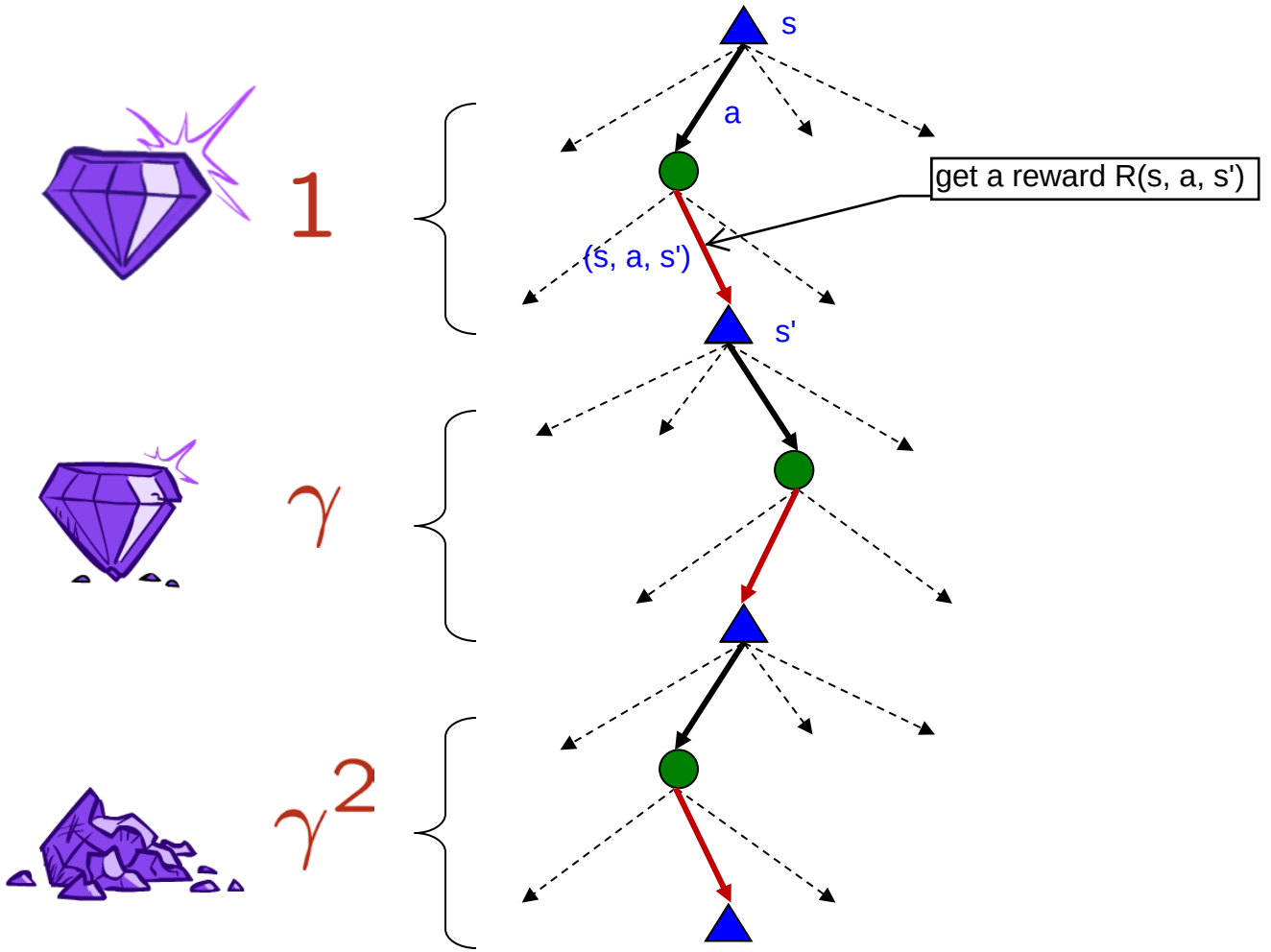
Worth Next Step

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:

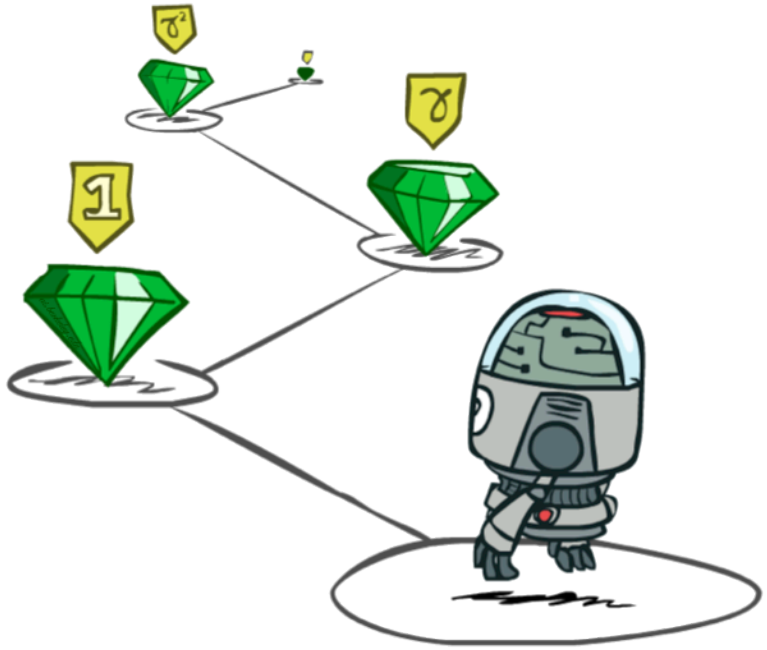
assume we like 'a' better than 'b'

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

$$\Updownarrow$$

if stationary, we like 'a' better than 'b' in the future too.

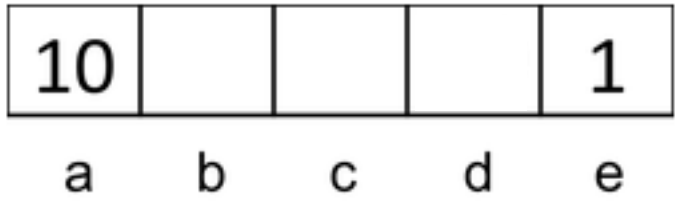
$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$



- 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?

Infinite Utilities?!

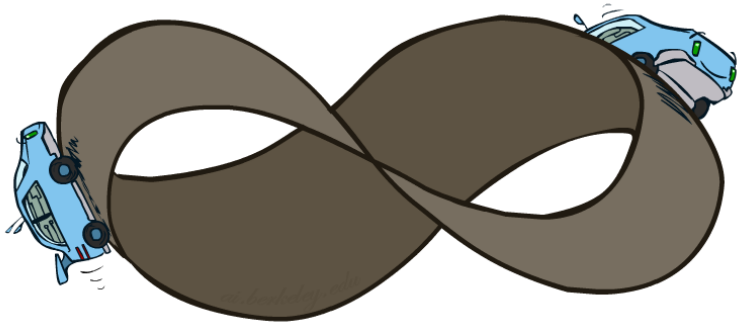
- Problem: What if the game lasts forever? Do we get infinite rewards?

  If the game lasts forever, it's hard for our agent to pick the "best", because we always could get infinite rewards. So how do we fix this issue?

- Solutions:

just declare that they end

  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)



  - Discounting: use $0 < \gamma < 1$

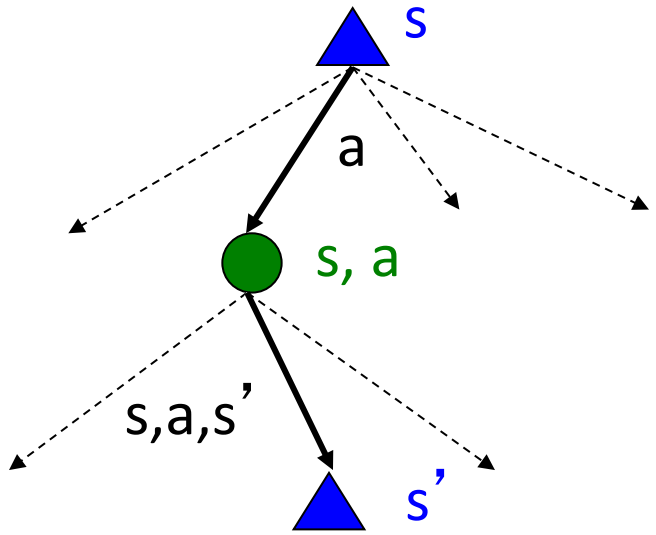
$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

    - 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)

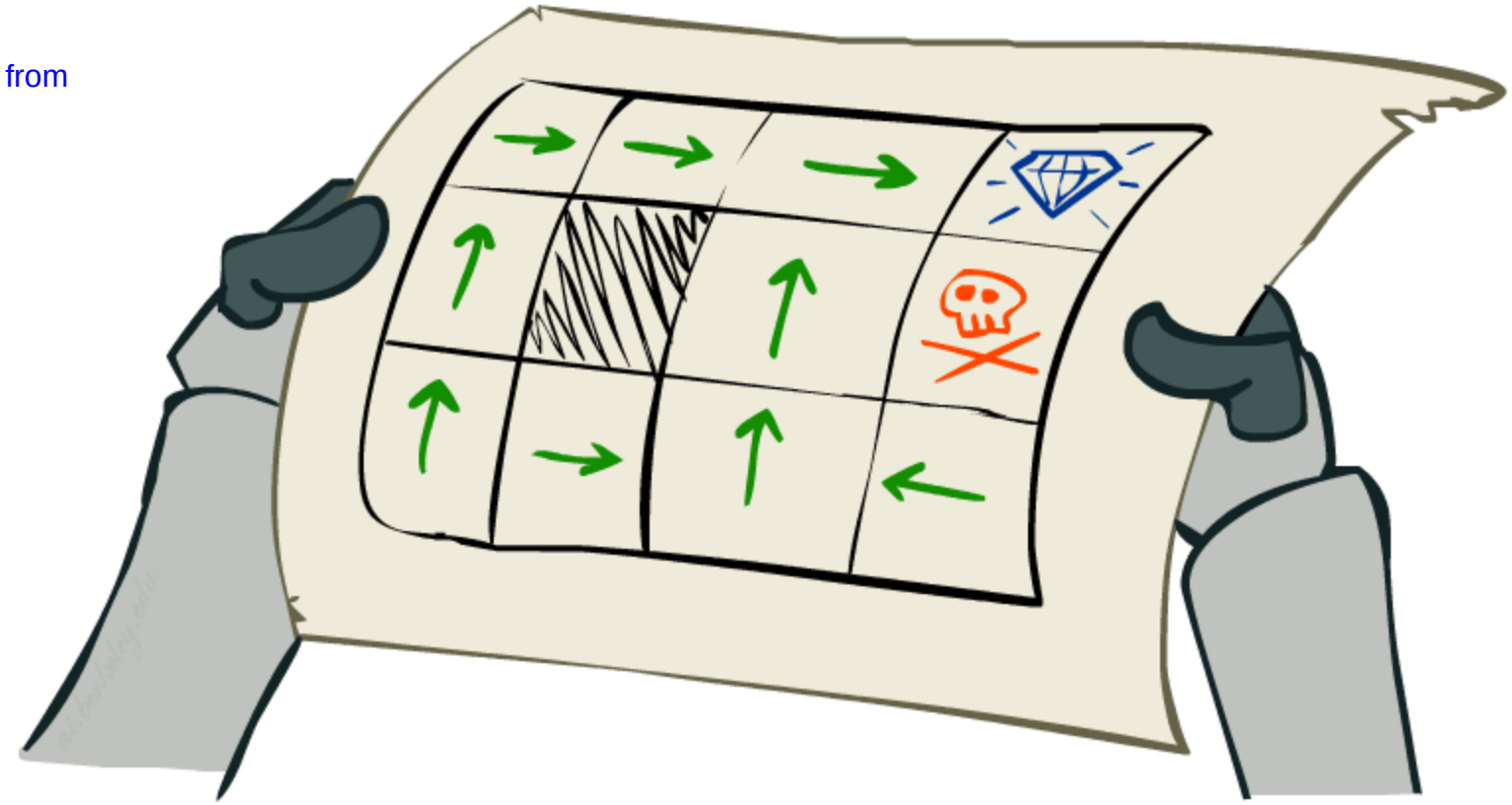


- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

Solving MDPs

The input for solving an MDP is an MDP. "Someone tells you here's the states, actions, transition probabilities, rewards".

The output is a policy, a mapping from each state to the optimal action.



Optimal Quantities

- The value (utility) of a state s:

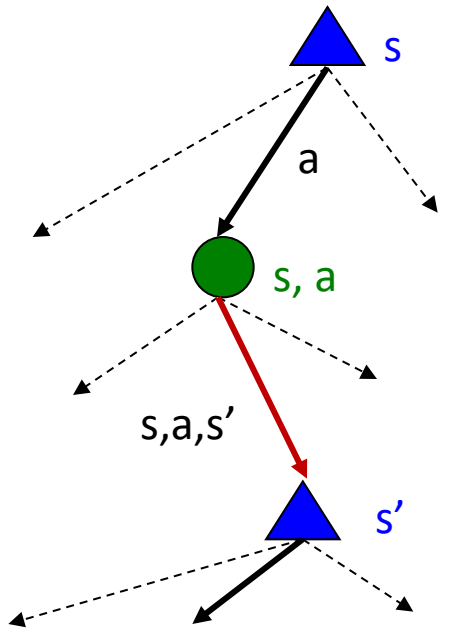
  $V^*(s)$ = expected utility starting in s and acting optimally

- 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



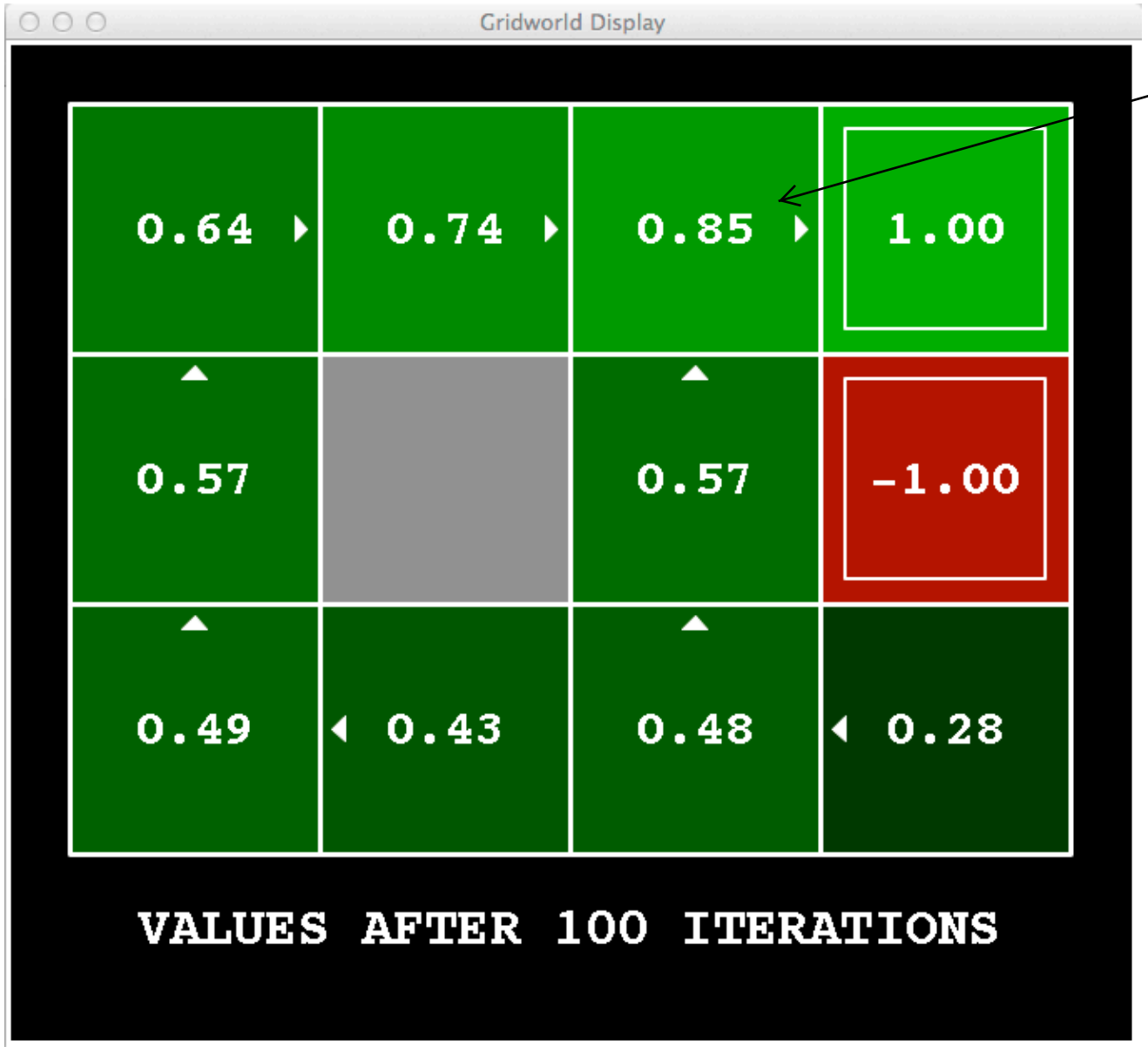
s is a *state*

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

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

[Demo – gridworld values (L8D4)]

Snapshot of Demo – Gridworld V Values

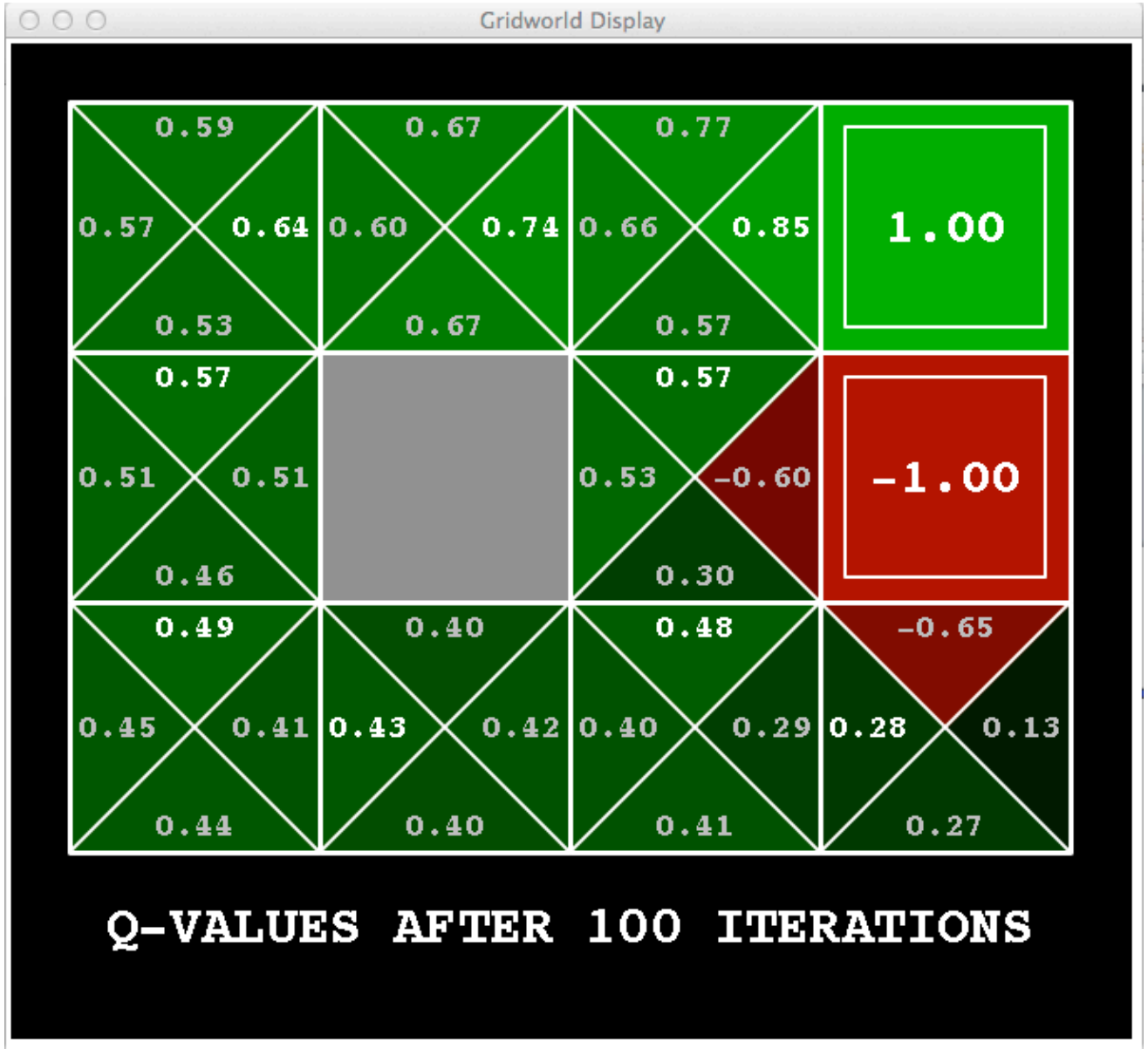


if noise = 0, then this would be 0.9 (due to the discount). But since there is noise, it's possible that we will try to go to the Goal tile, but will actually not go that way.

Noise = 0.2
Discount = 0.9
Living reward = 0

Snapshot of Demo – Gridworld Q Values

The max Q-state is equal to the max value of the state from the previous slide!



Noise = 0.2
Discount = 0.9
Living reward = 0

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:

weighted average over all possible transitions for (s, a)

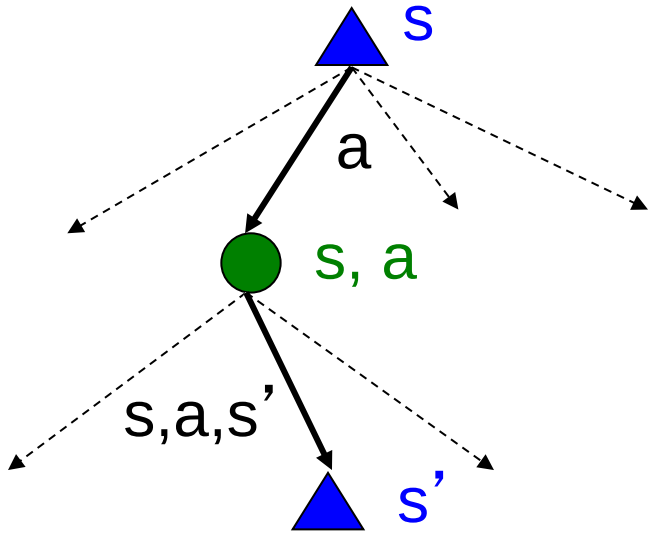
Core equations!

These are the Bellman equations!

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

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

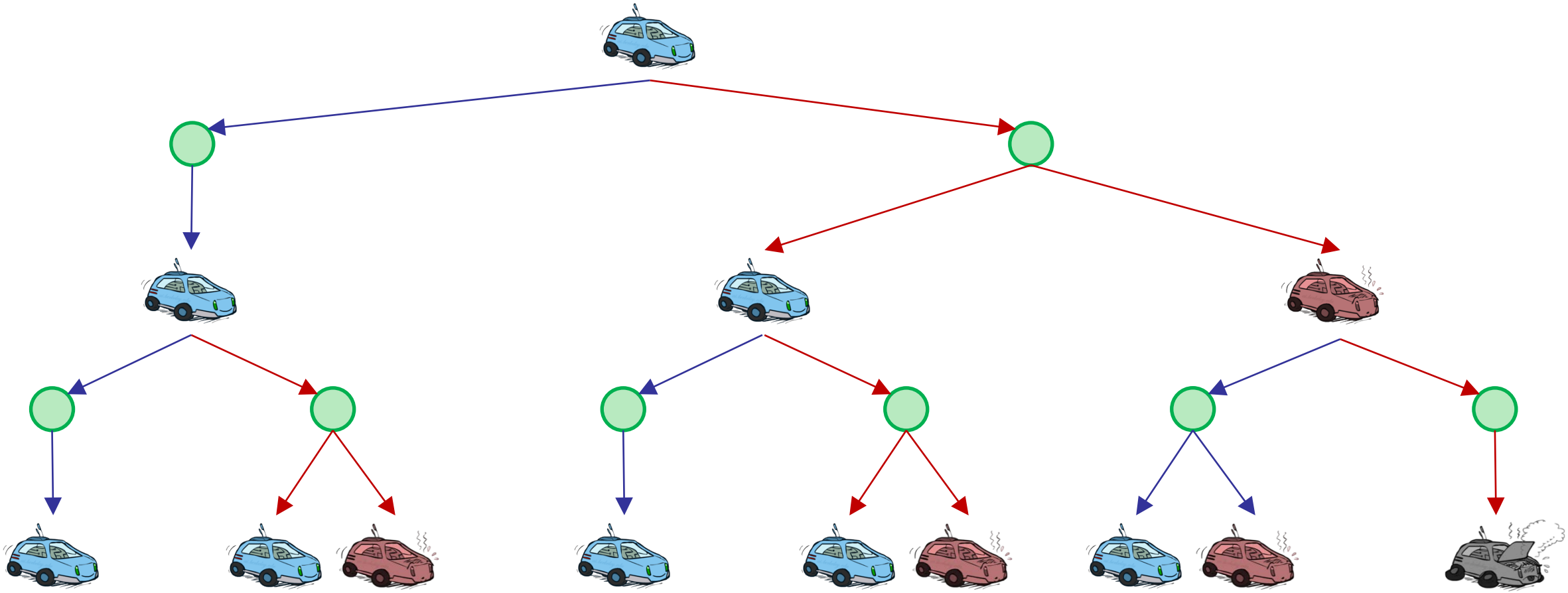
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$



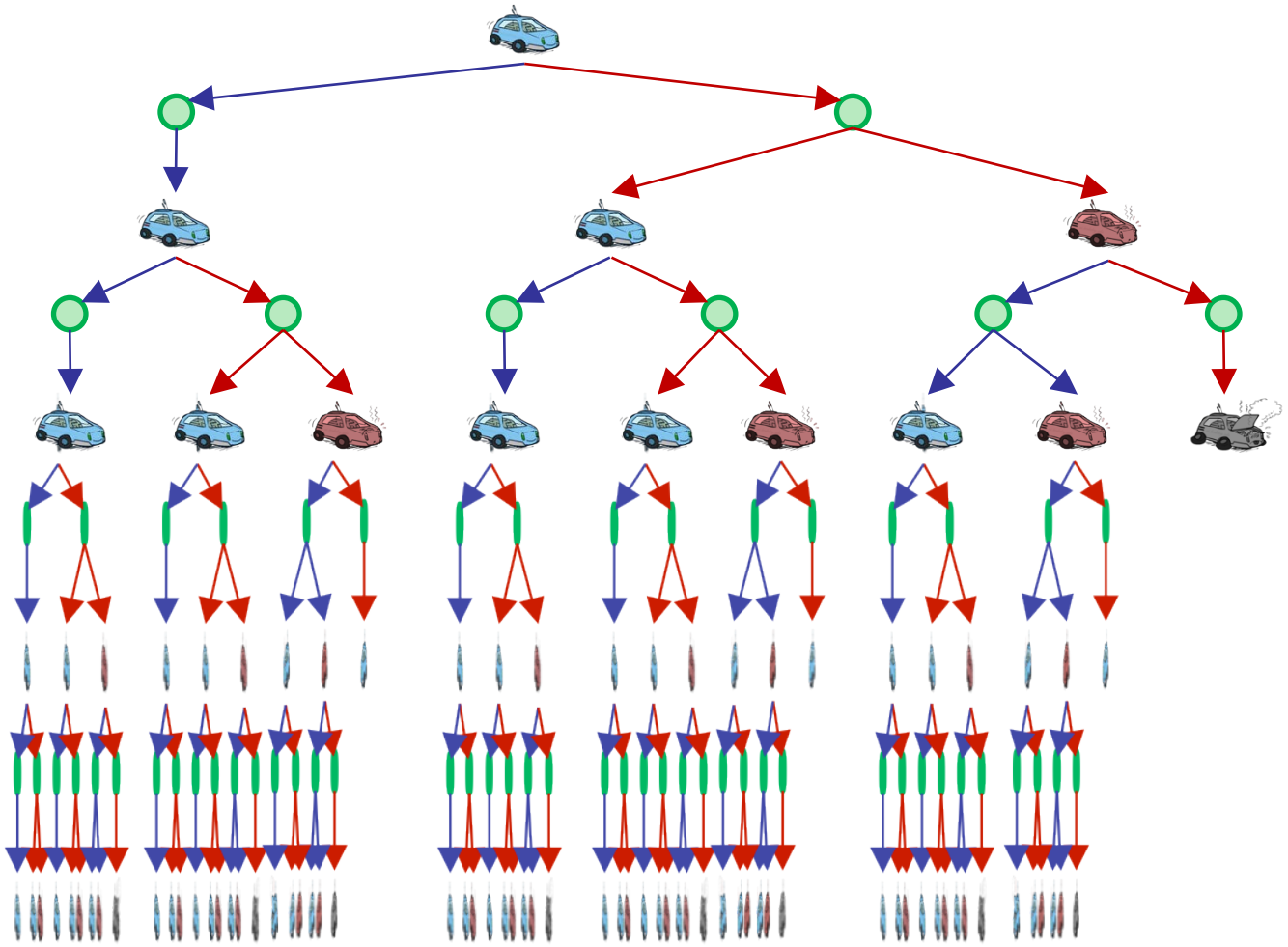
You can define the optimal value of states in terms of the optimal value of q-states.

And you can define the optimal value of q-states in terms of the optimal value of states

Racing Search Tree



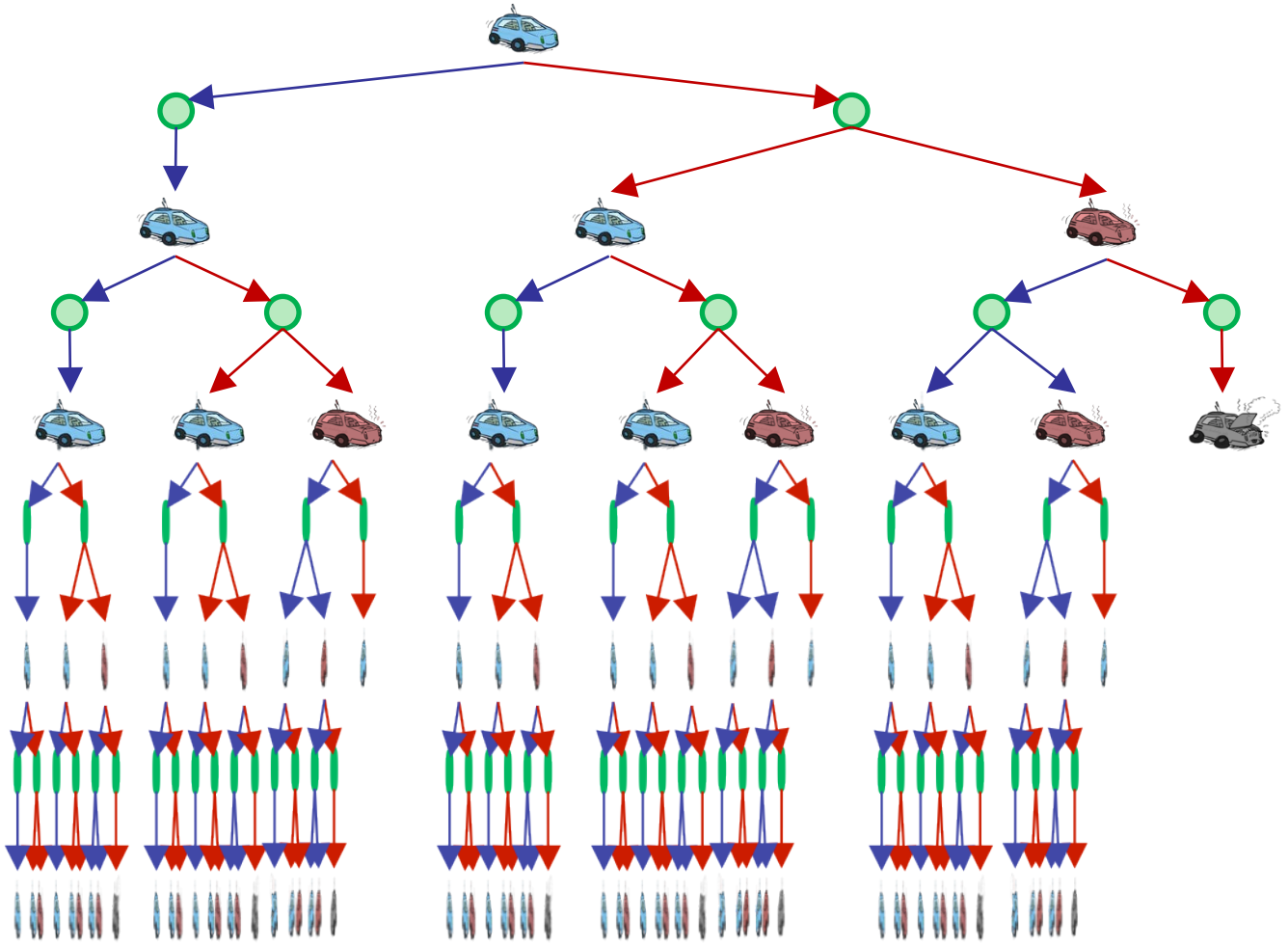
Racing Search Tree



Racing Search Tree

To avoid these issues, we'll use an algorithm called "Value Iteration"

- 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 $\gamma < 1$

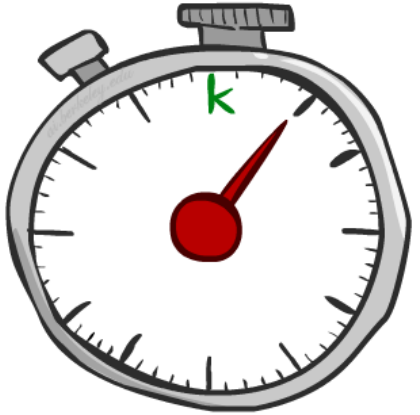


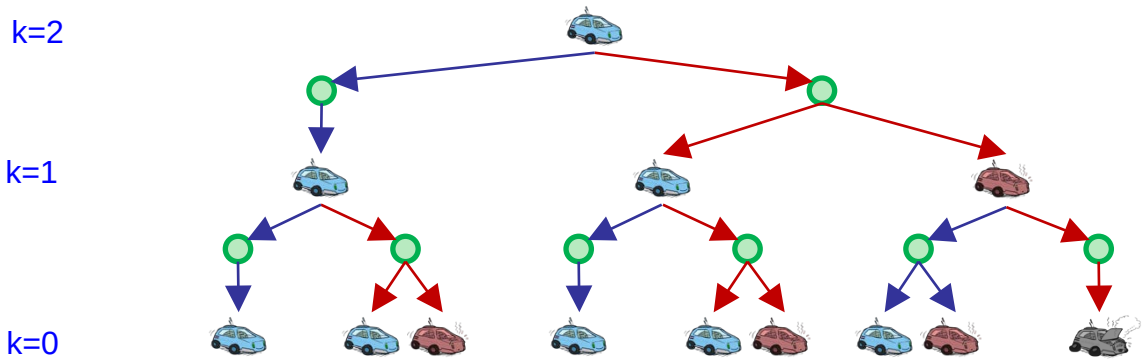
Time-Limited Values

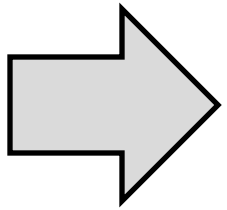
- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps

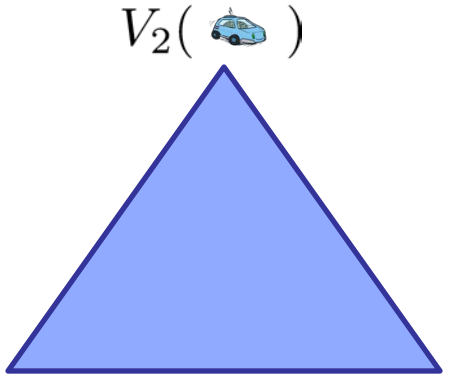
so now the tree is no longer infinite

  - Equivalently, it's what a depth-k expectimax would give from s







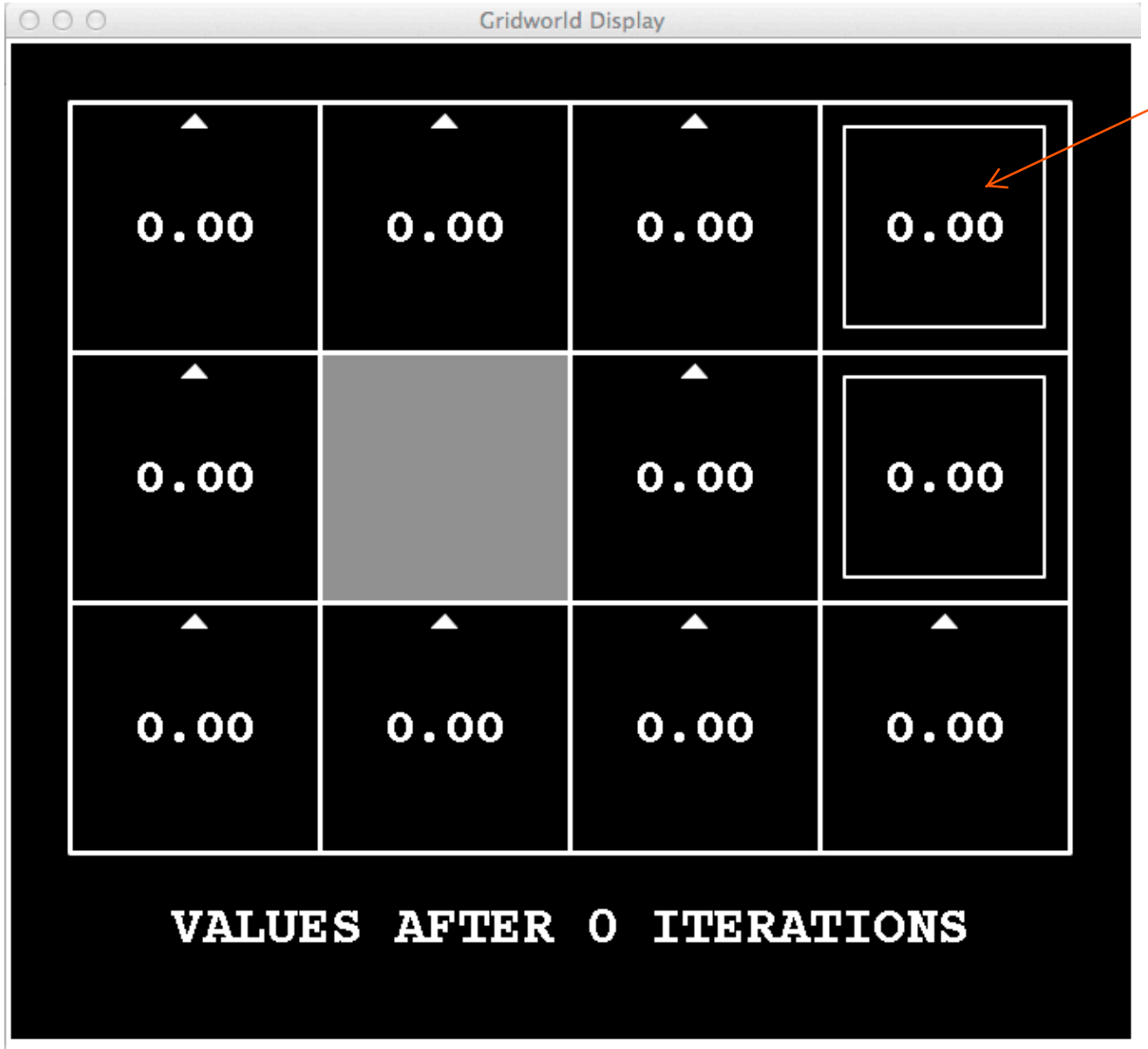


using Bellman's Equation, we don't go top down, but rather we go bottom up. That means we start at the "end" state k=0, then slowly work our way to k=2

[Demo – time-limited values (L8D6)]

k=0

if we have zero time steps (aka rewards), we get 0 rewards

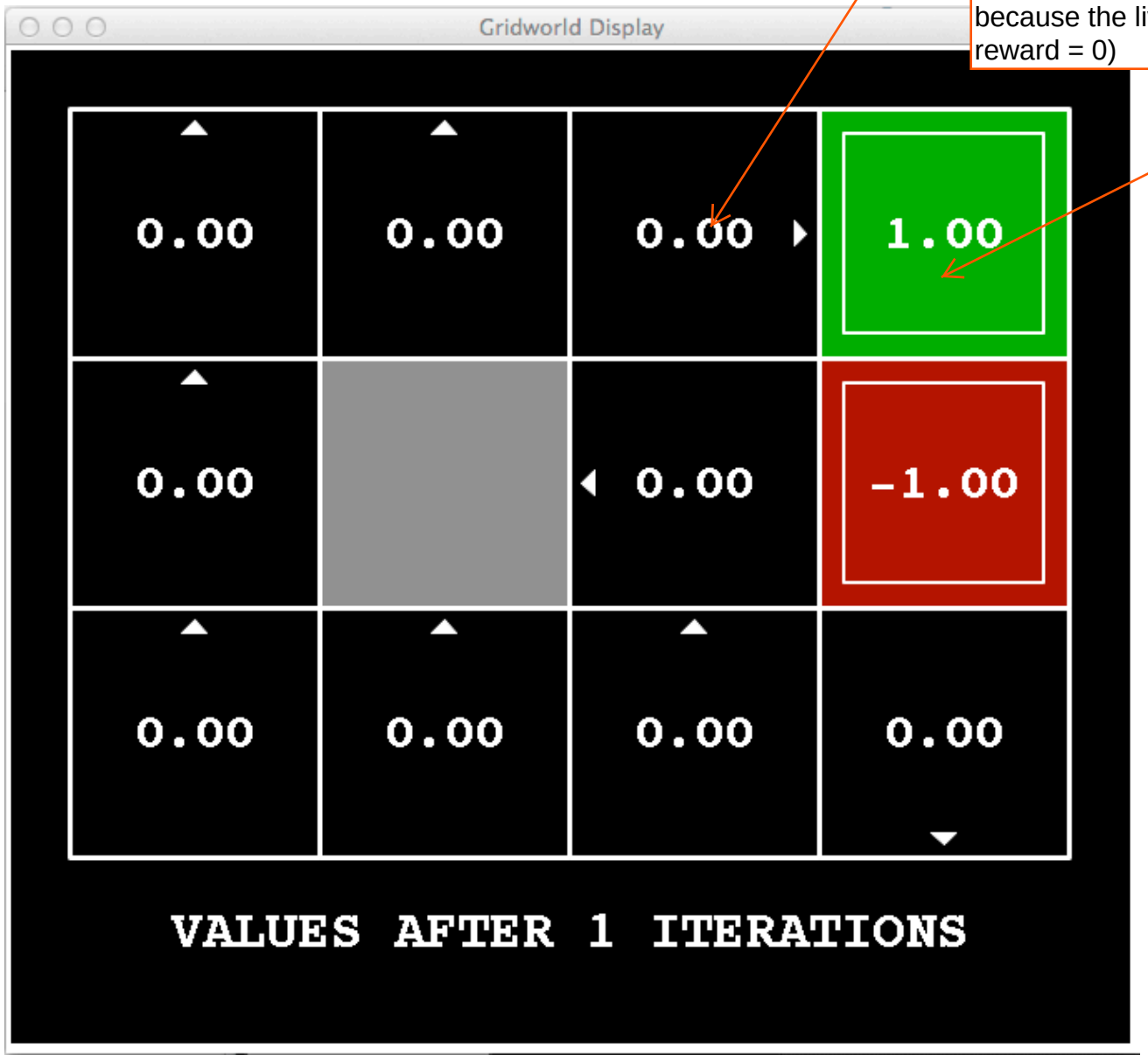


this is zero because we need to take another action to get the +1 reward.

This was just how Gridworld was defined.

Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



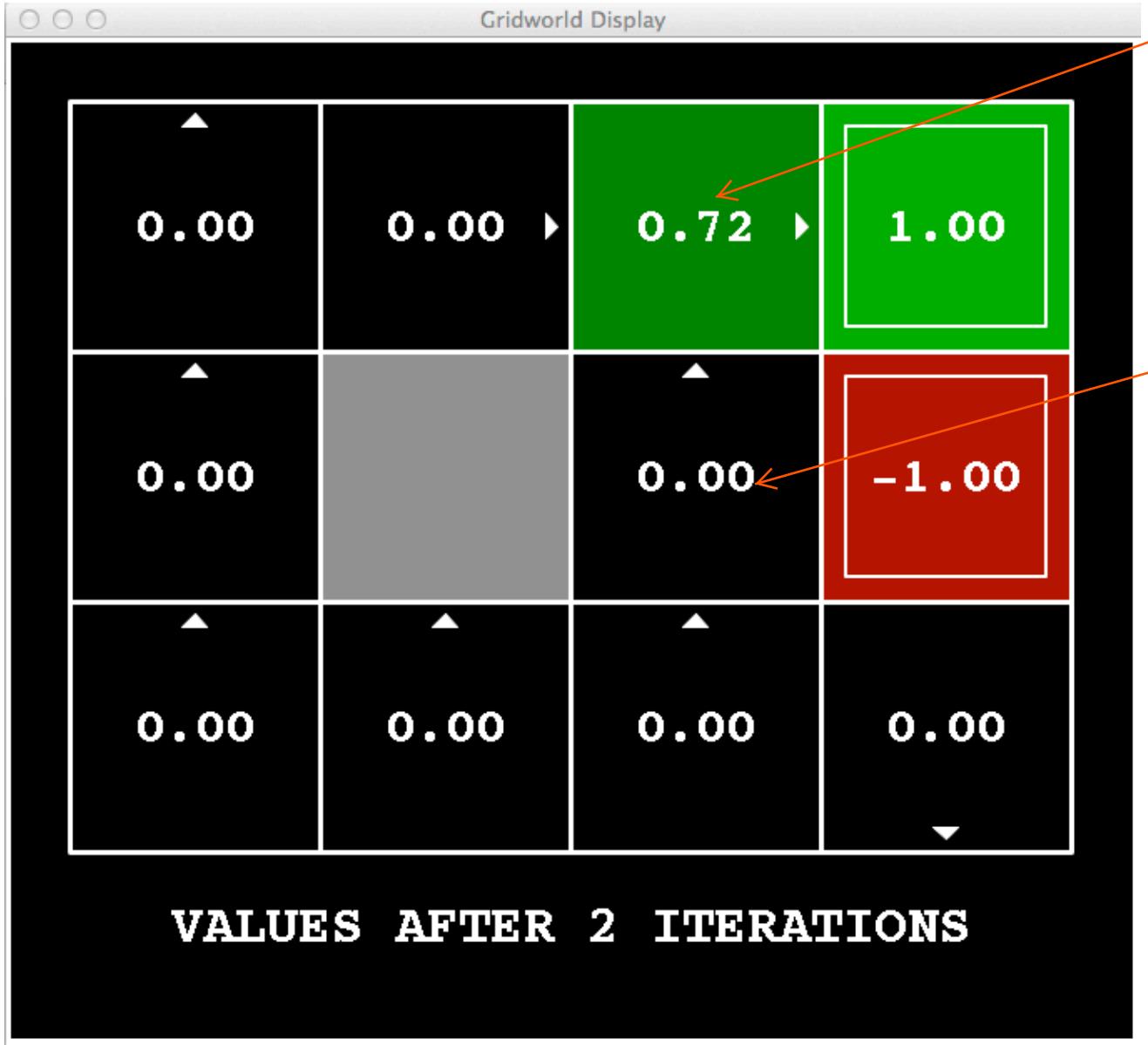
We could move to the goal square, but we won't have time to exit that square, so we still get 0 reward. (this is because the living reward = 0)

Now, with one time step, we can exit the square and get + 1

Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

the policy is changing a lot



Now we could actually exit. We could also accidentally go down, then try to go back up, then run out of time. With k=2, we can't actually fall into the pit

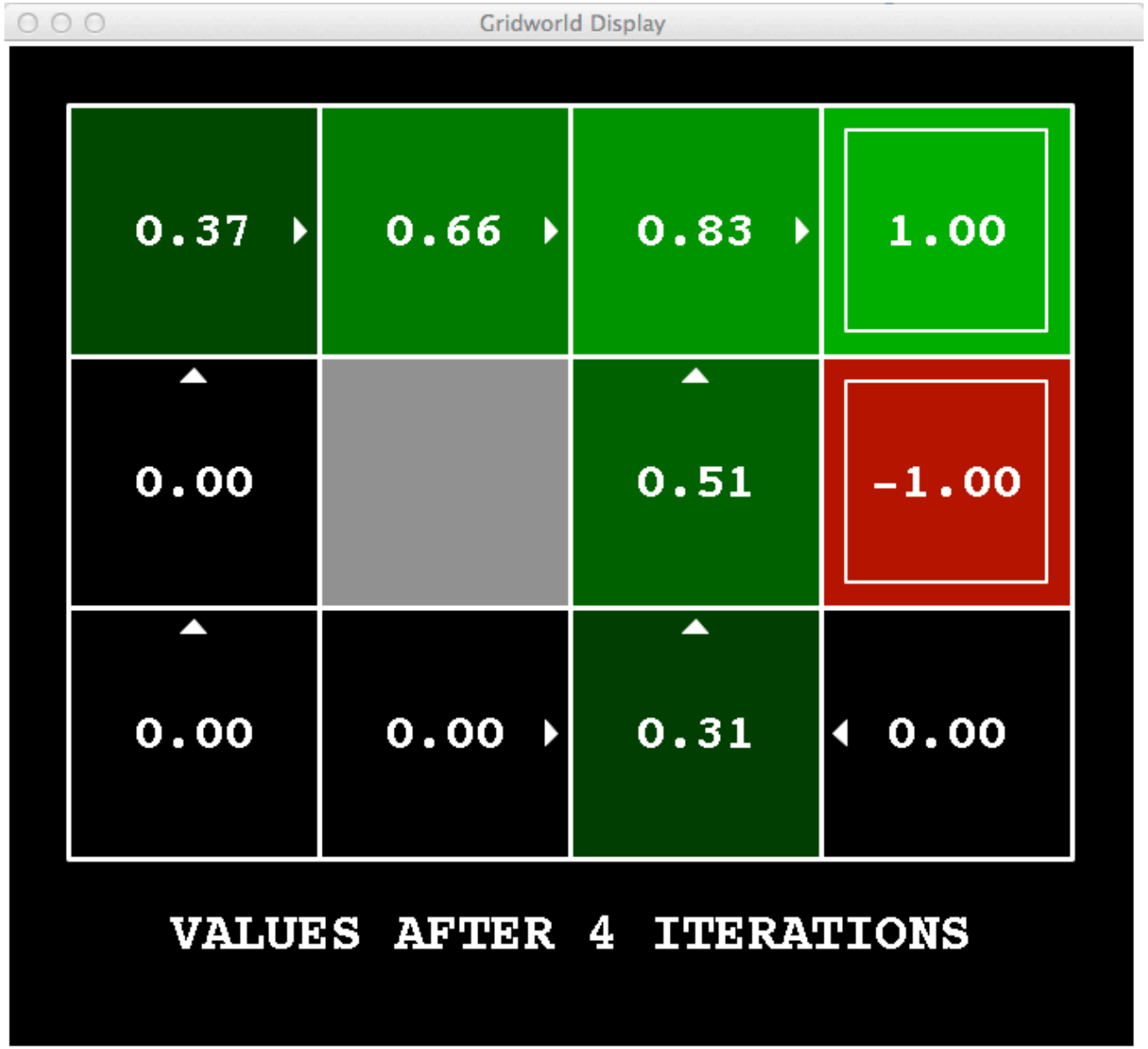
You can take steps to guarantee you get 0

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

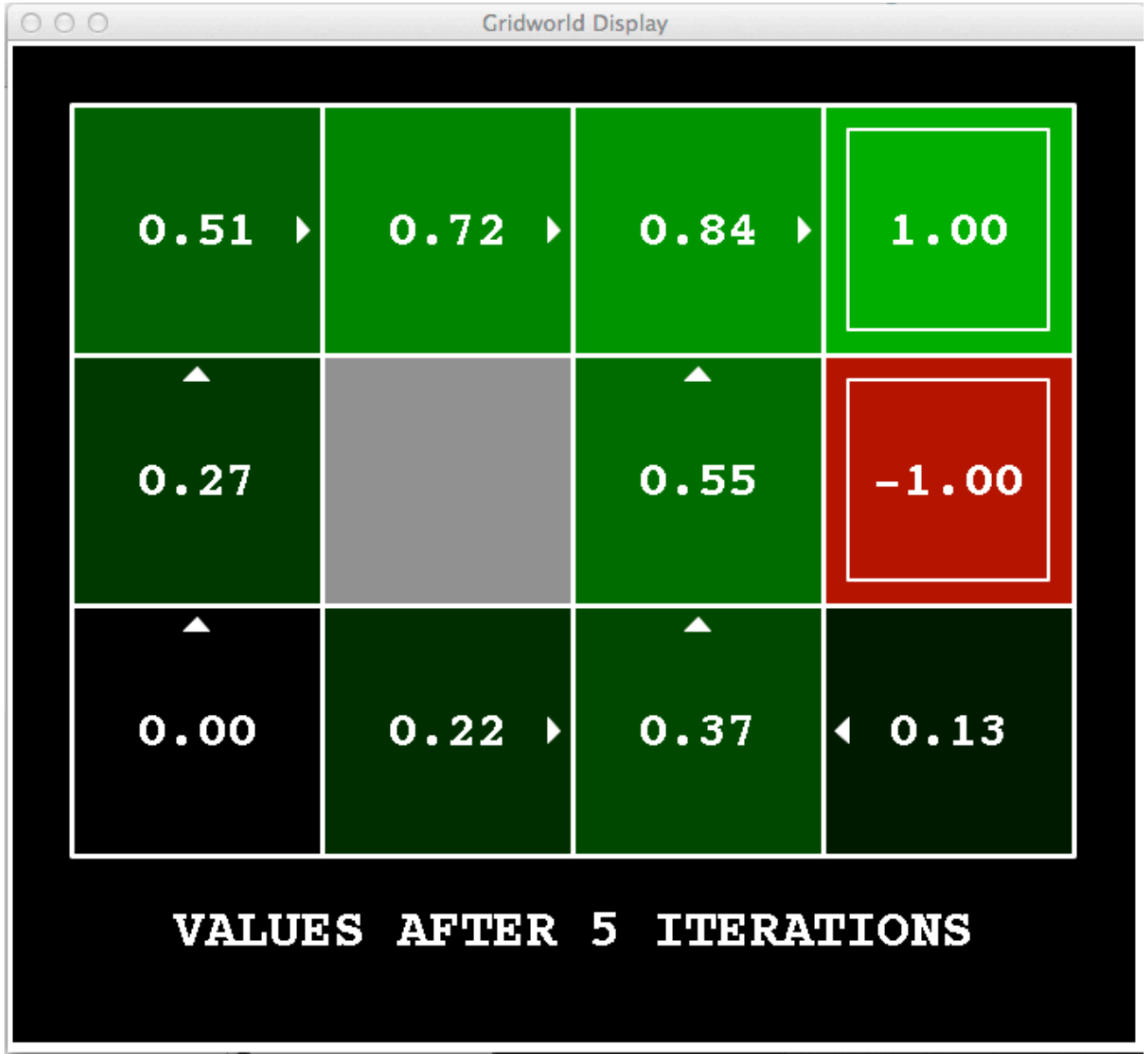
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



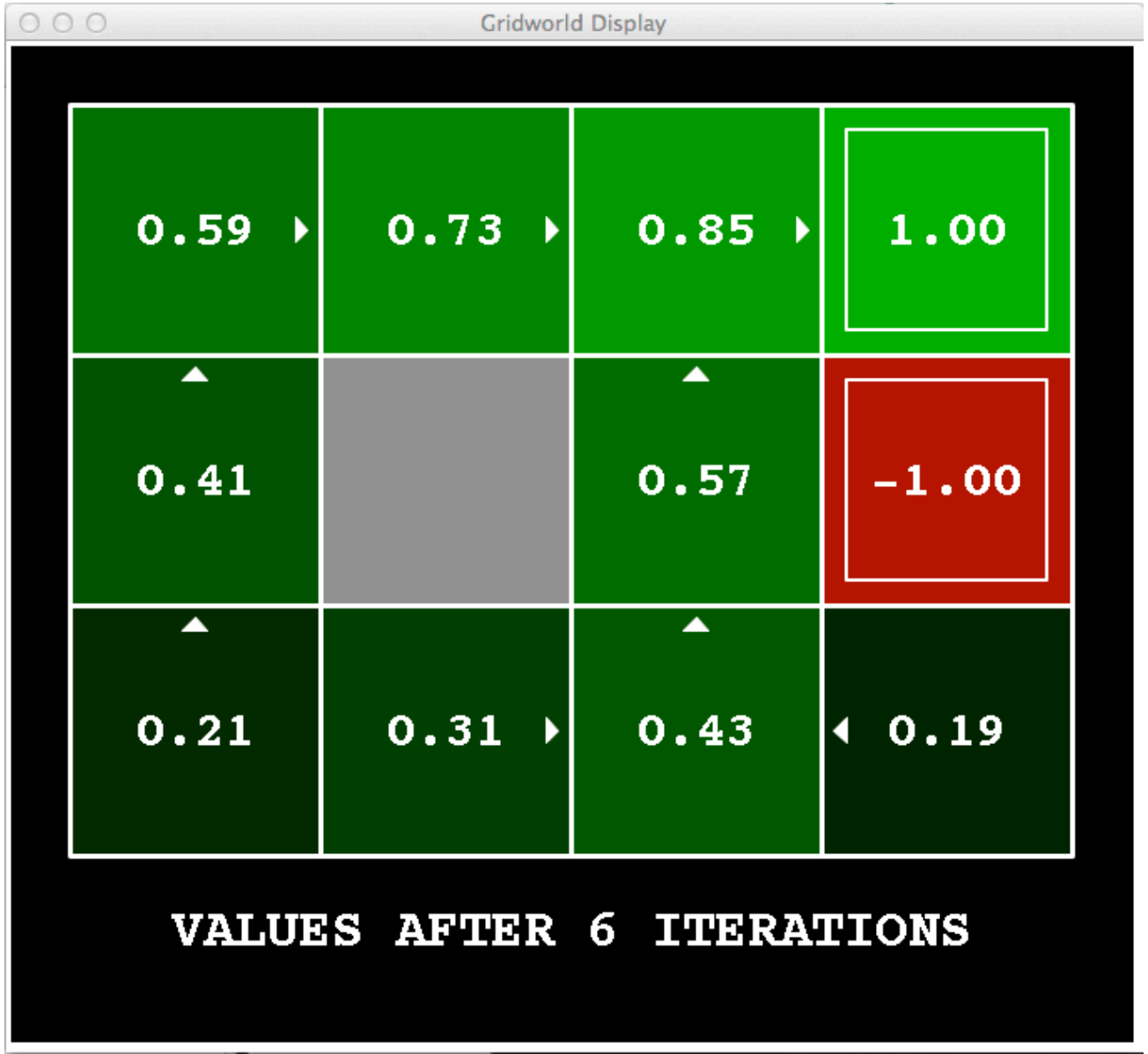
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



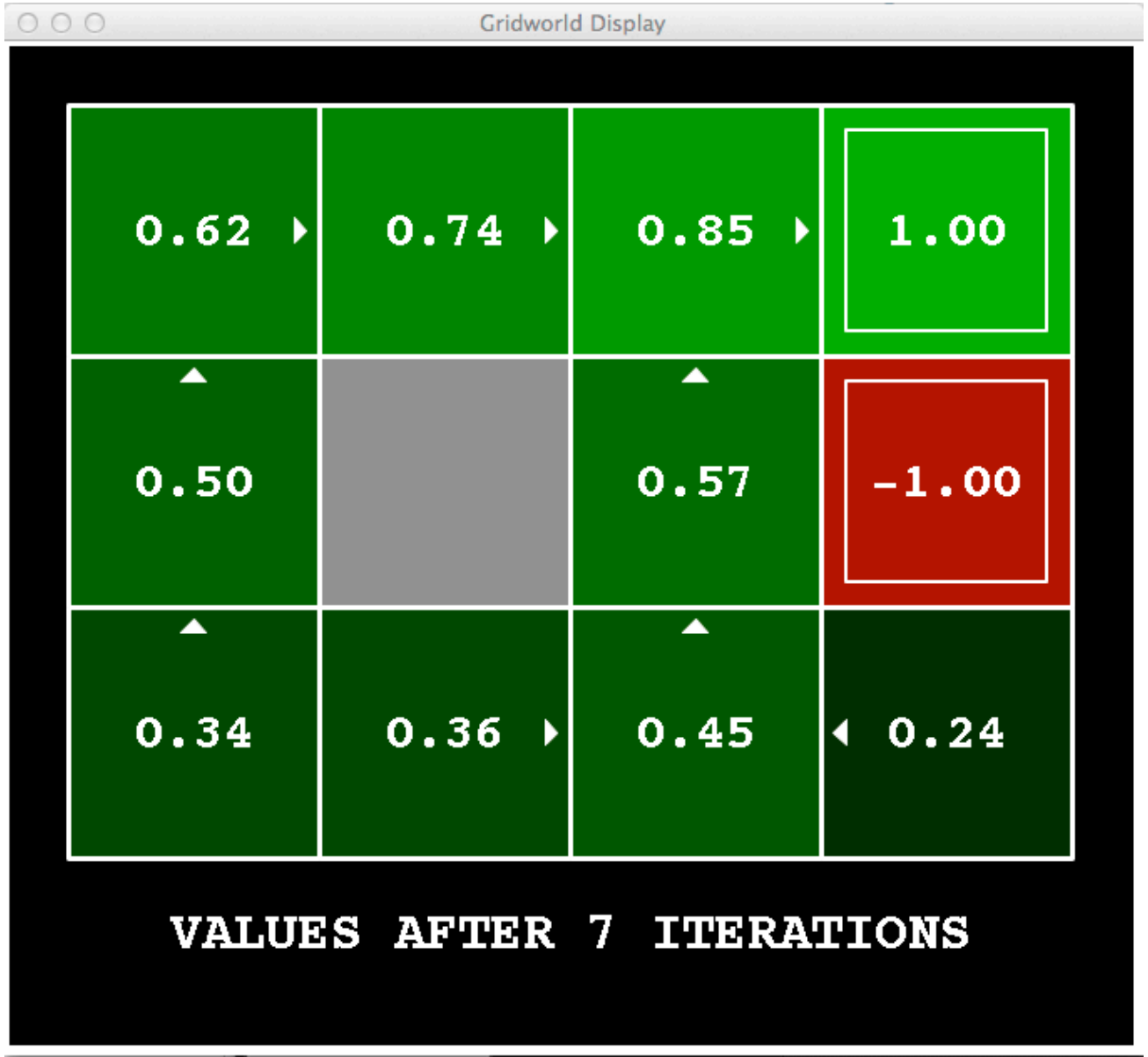
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



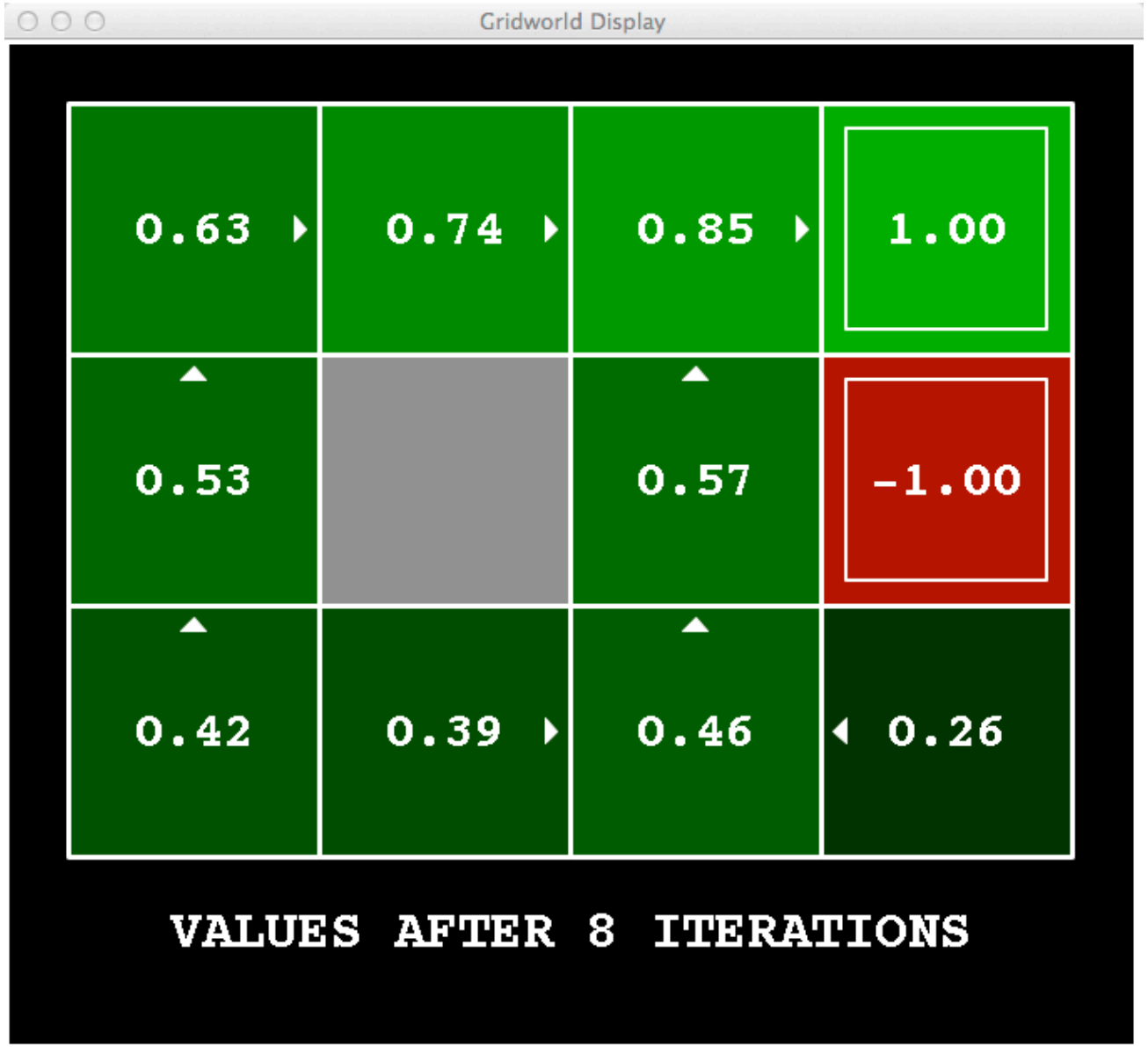
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



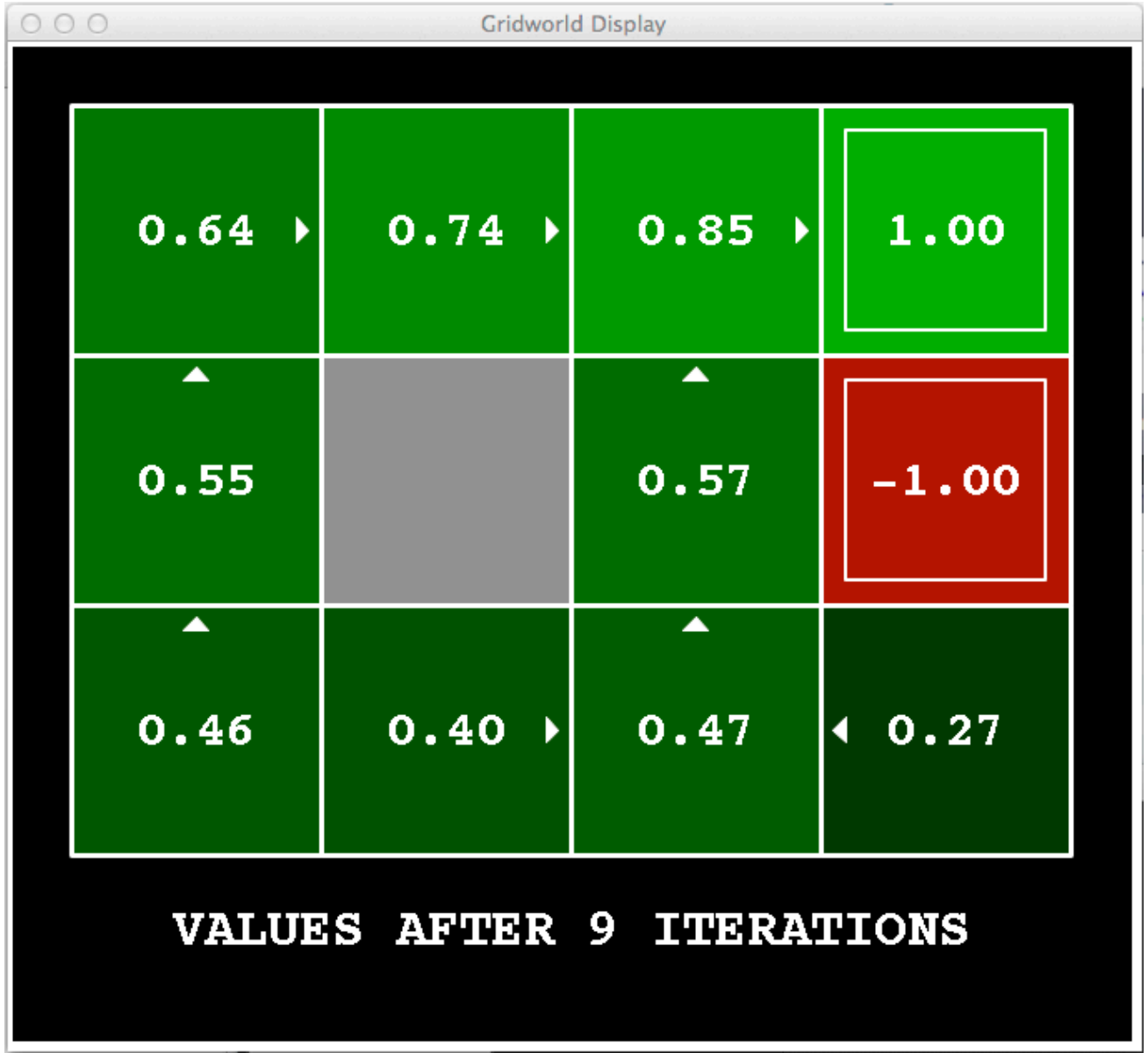
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



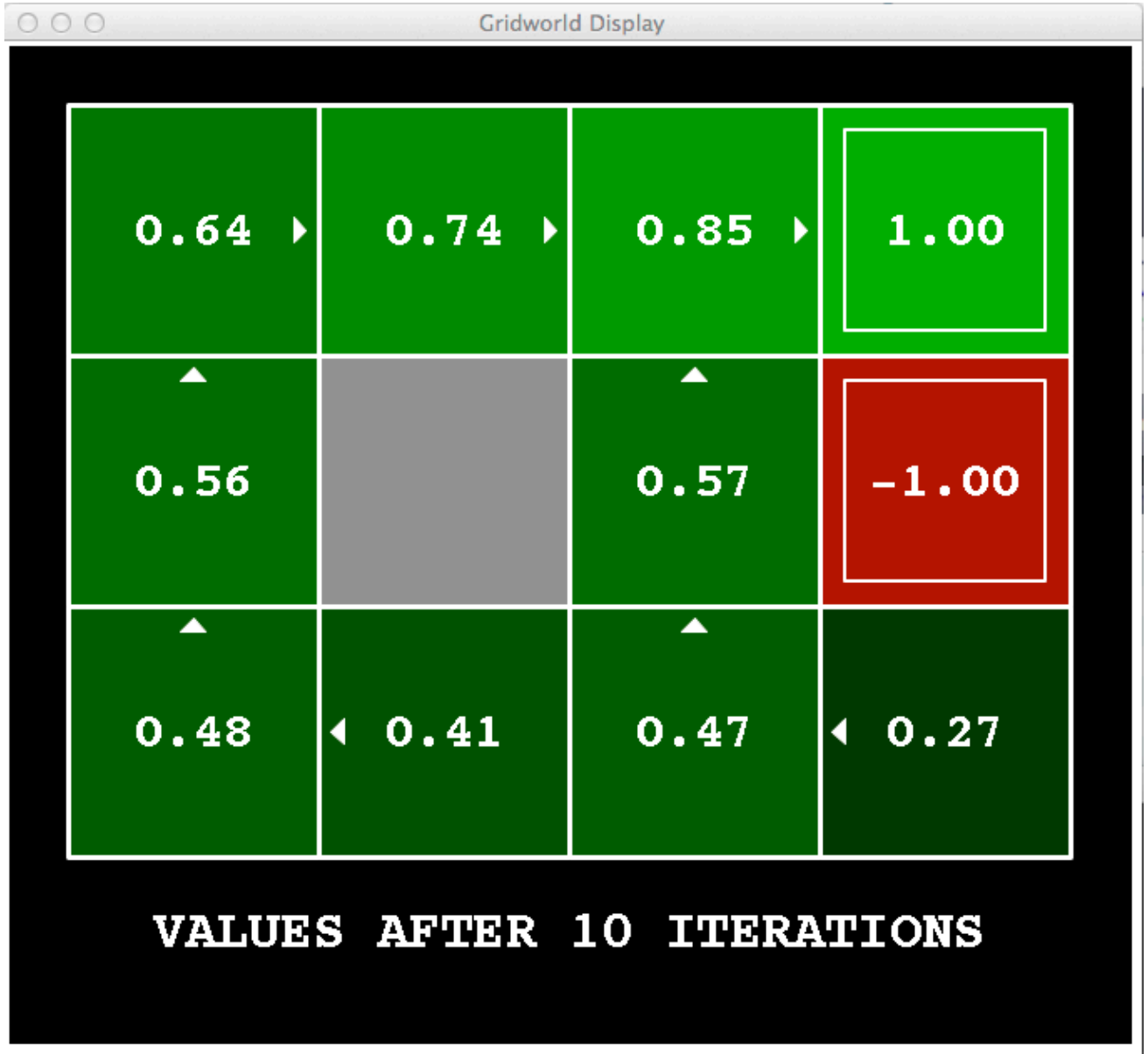
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



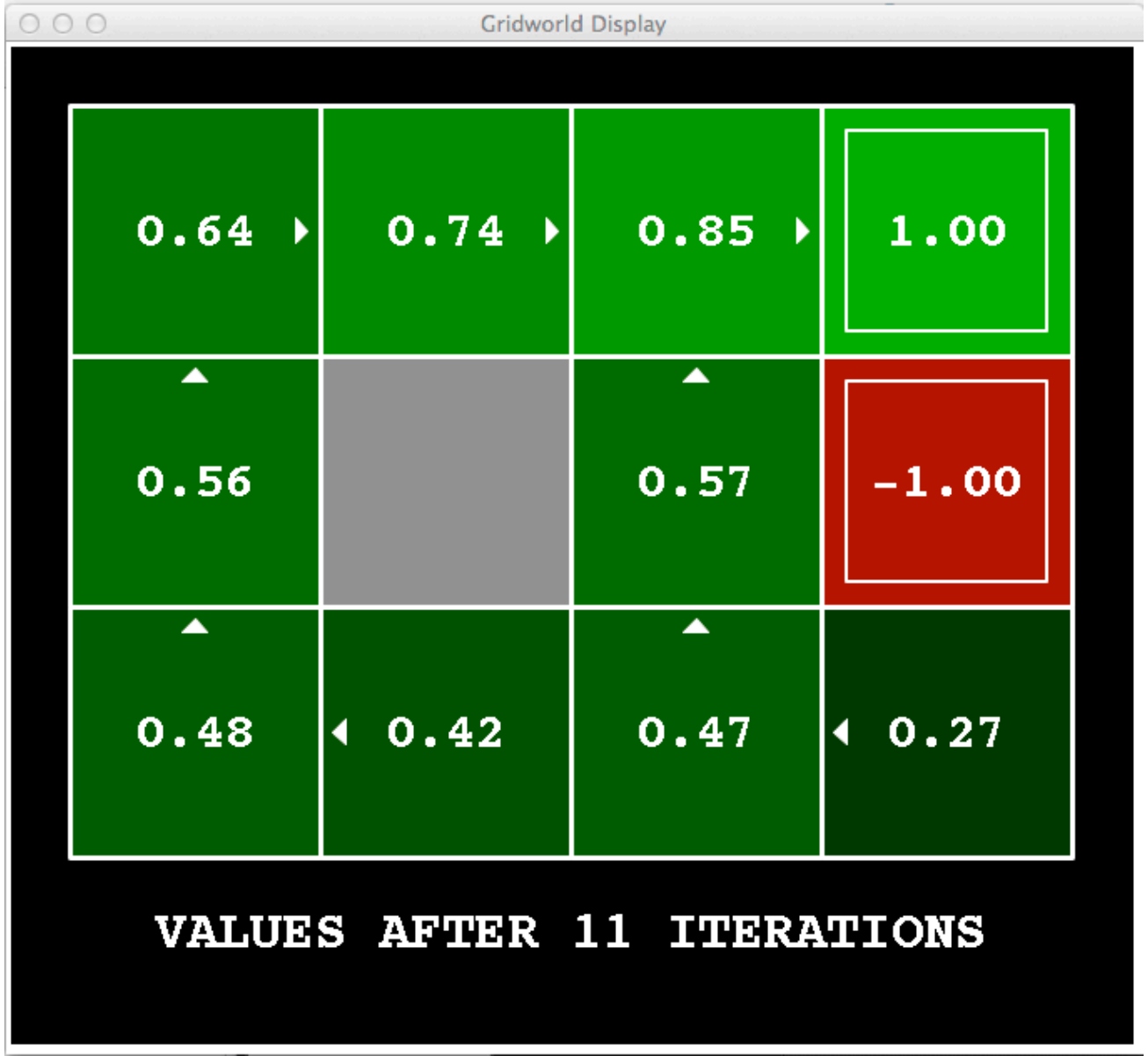
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



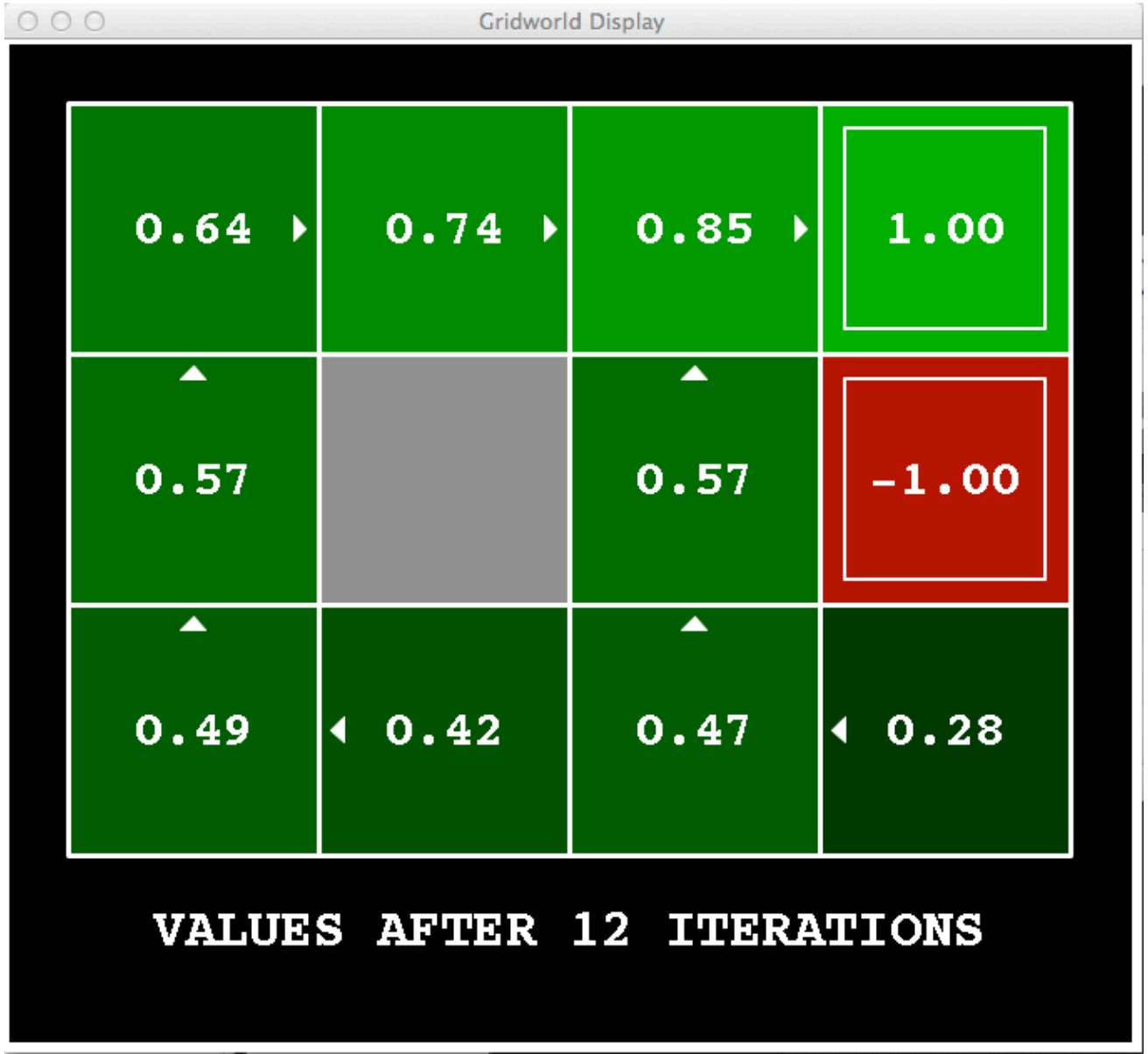
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

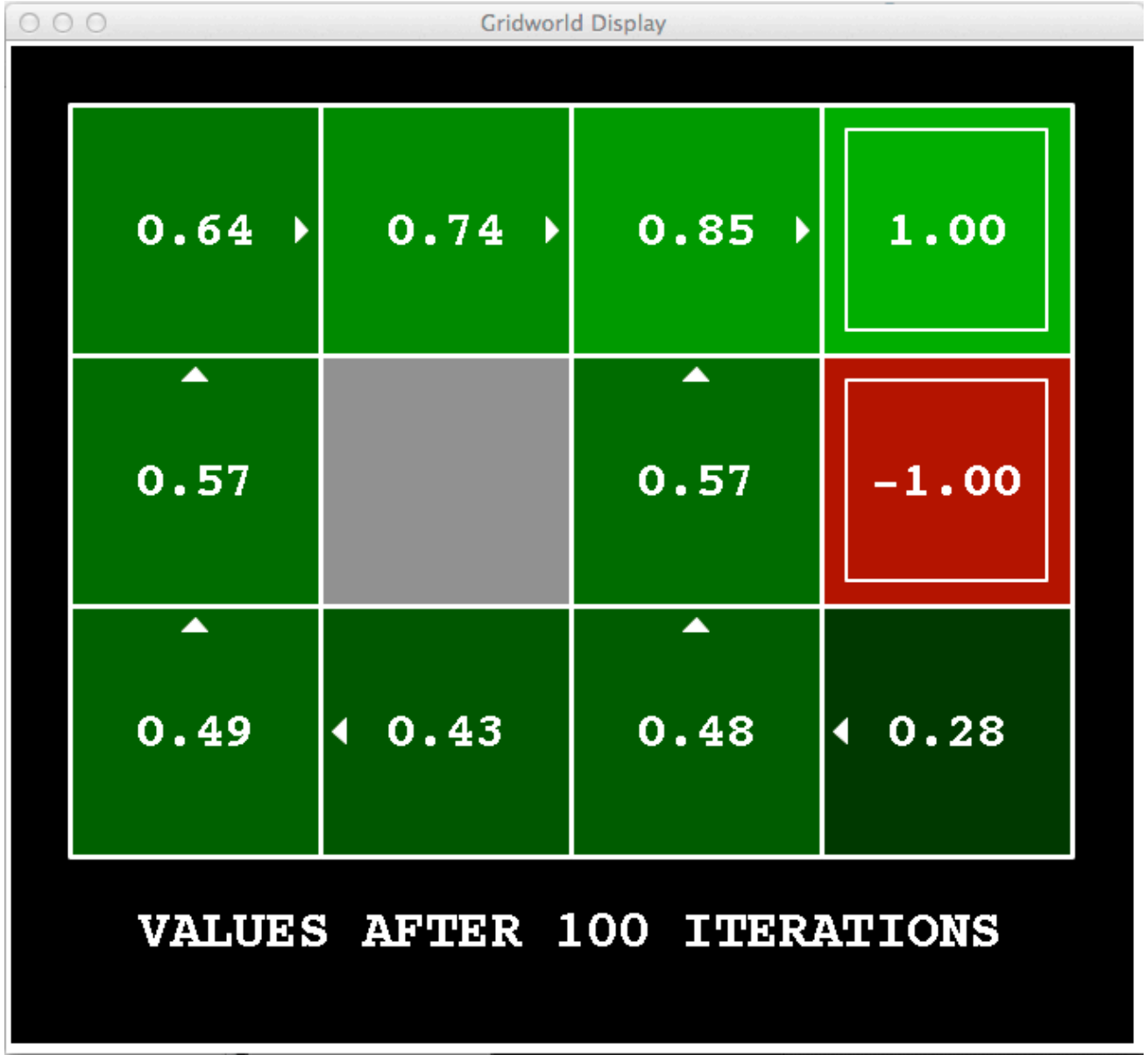
k=12



notice that the policy isn't really changing anymore, but the numbers are slightly changing

Noise = 0.2
Discount = 0.9
Living reward = 0

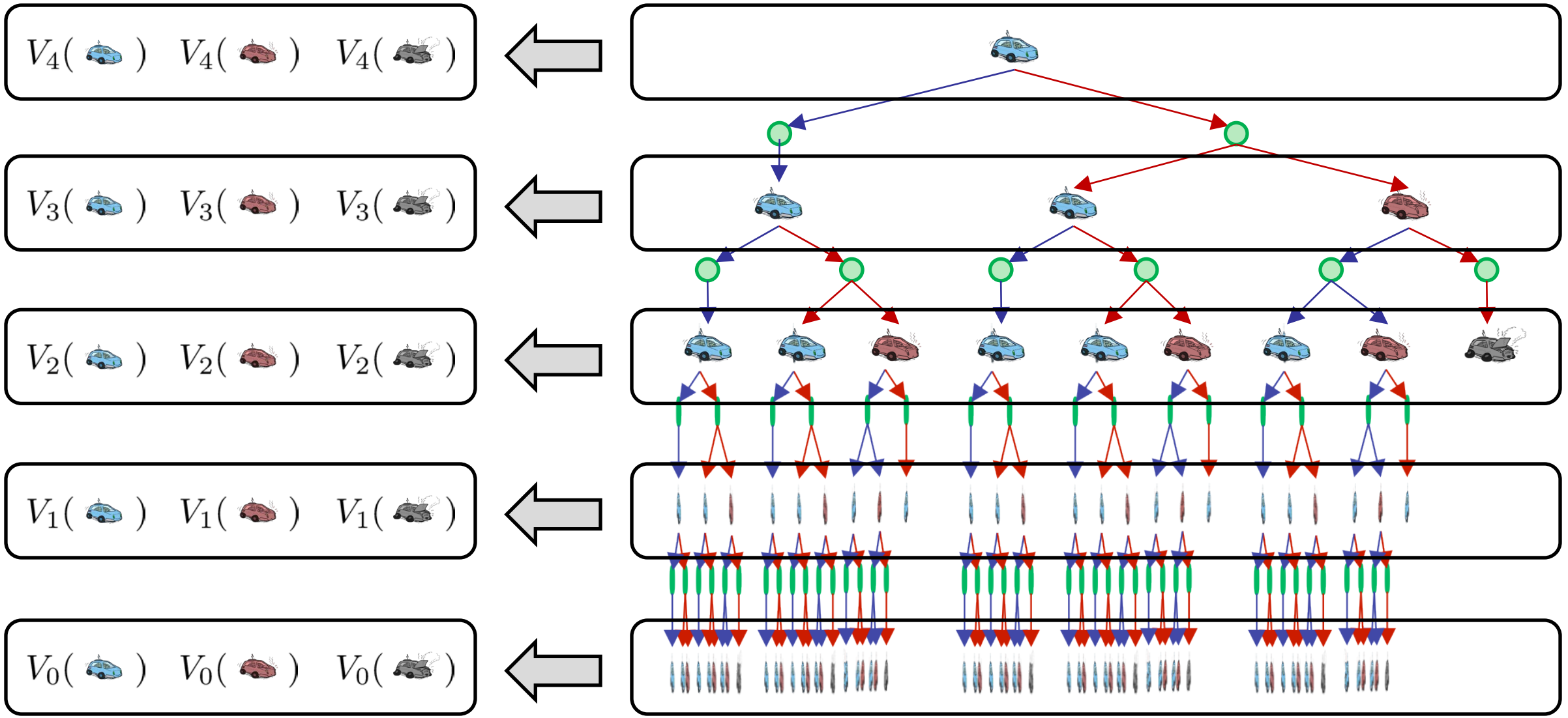
k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

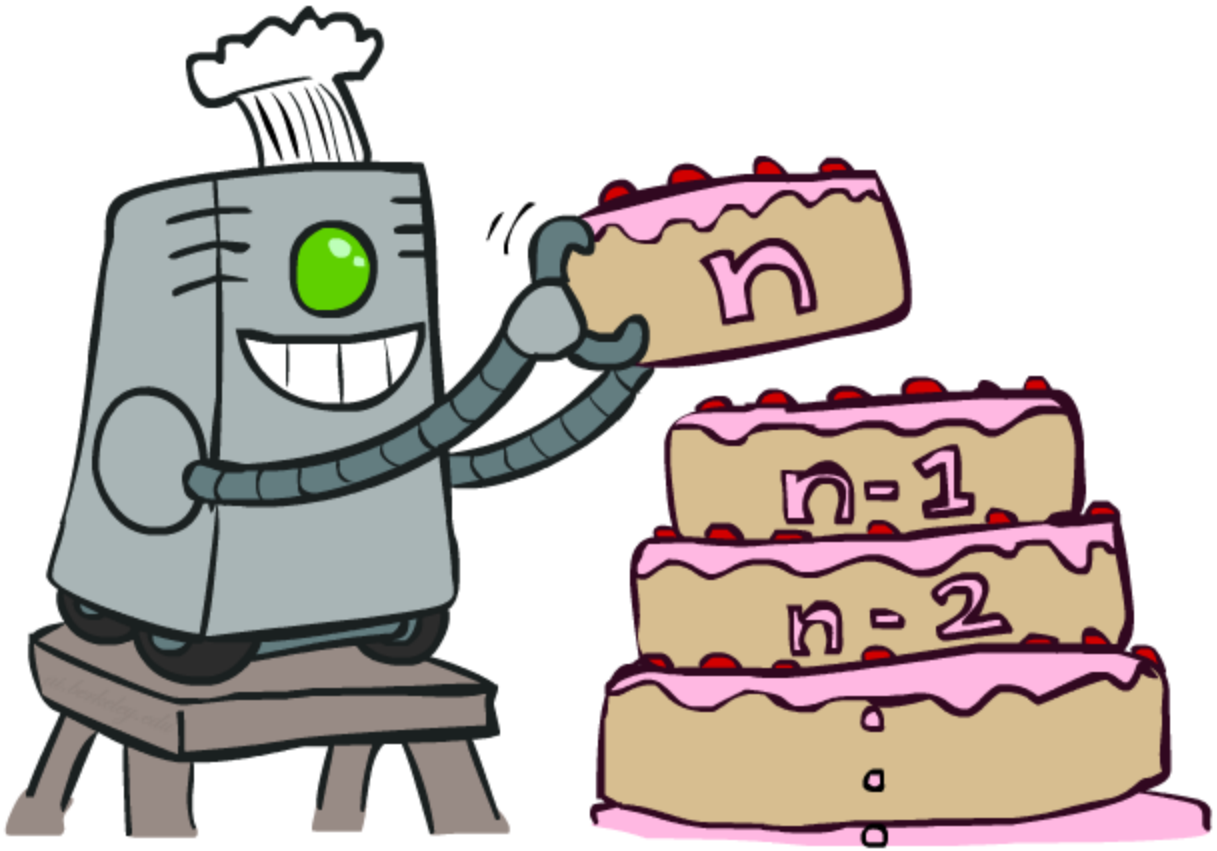
we can summarize all of this complexity in the tree in the simple Vk(s)

Computing Time-Limited Values



Value Iteration

start by saying "what can I accomplish in 0 time steps", then "what can I accomplish in 1 time step", and so on until we converge



compute average reward for optimal play with 0 time steps

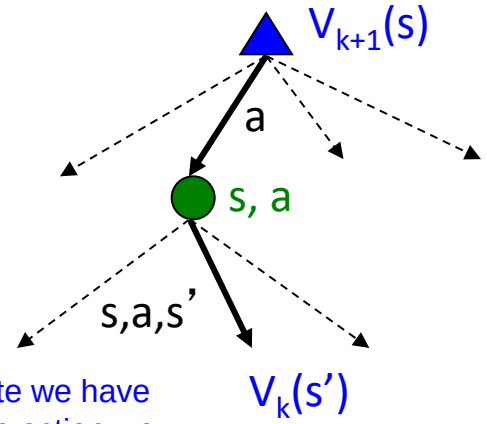
this only works when the number of states is manageable and can be enumerated

Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

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

compute average reward for optimal play with k+1 time steps left



- Repeat until convergence

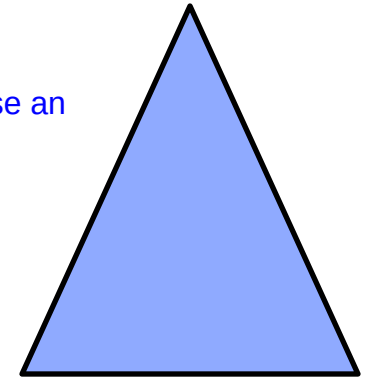
we have to look through each state S, and for each state we have to look at each possible action A, then for each possible action we have to look at each possible S'. Thus we get S * A * S

In practice we don't typically have to look at the all S' because an action will result in a limited number of possible S'

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

Is this complexity good? Well, if you have a lot of states, then no.

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



Value Iteration

Two possible actions: Go fast or go Slow
Action-Slow: I get +1, then I'm guaranteed to land at Cool again at S'. Then the next time step V1(cool), I'll take the optimal action and get 2. So in total I'll get +3 for going slow.

Action-Fast: I get +2, but then I could either land in Cool or Warm. If I land in Cool I'll do the optimal V1(cool) = 2. This would give me a total of 2 + 2 = 4
If I land in warm, I'll do the optimal V1(warm) = 1, then I'll get 2 + 1 = 3
I have a 50% chance of being in either state, my expected reward is 0.5(4) + 0.5(3) = 3.5

Since 3.5 > 2, V2(Cool) = 3.5 and my policy will tell me to go fast!

V_2 | 3.5 | 2.5 | 0

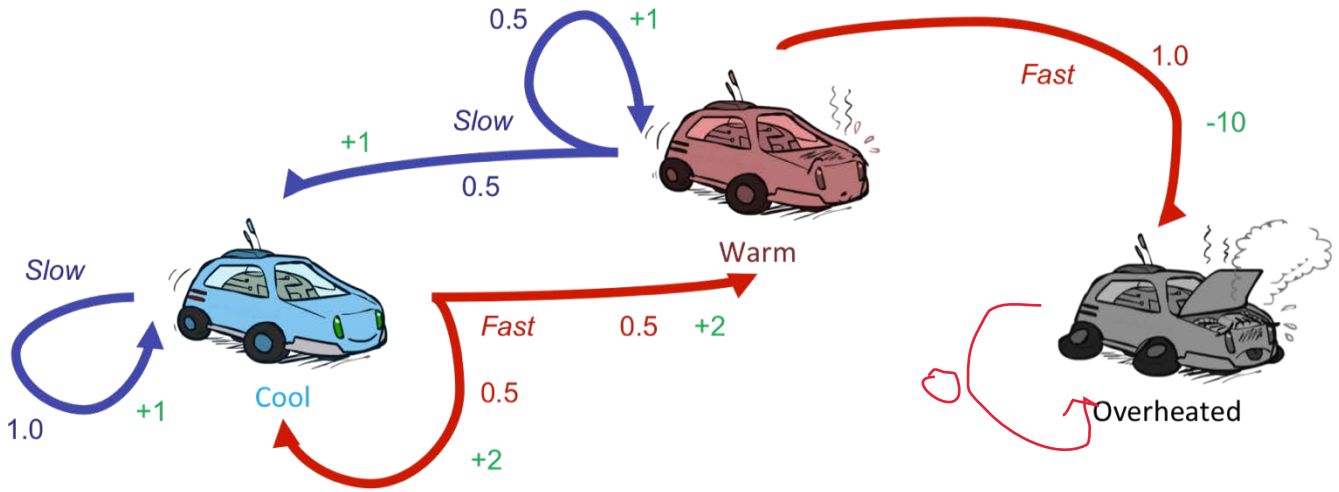
only one timestep left so go for it! go fast and get +2

V_1 | 2 | 1 | 0

you only get -10 as you transition to this state.

V_0 | 0 | 0 | 0

with zero time steps you can only get 0 rewards



Assume no discount!

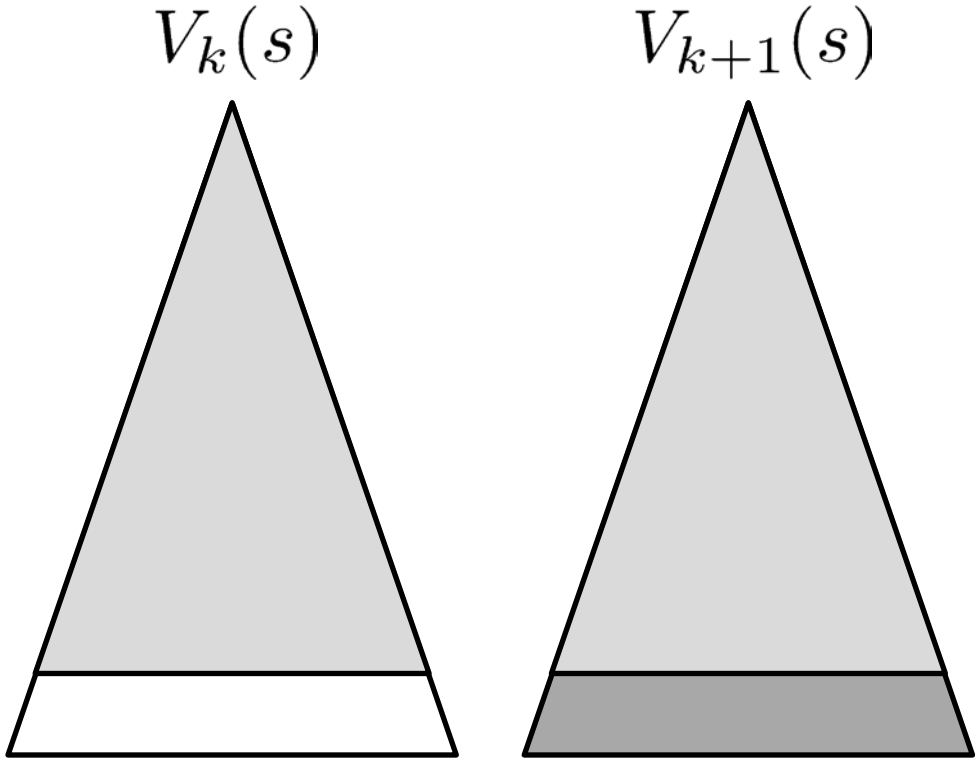
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Since we aren't discounting, these won't converge. We need to discount for it to converge!

Did not get to this slide in the MDP 1 lecture.

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



Next Time: Policy-Based Methods