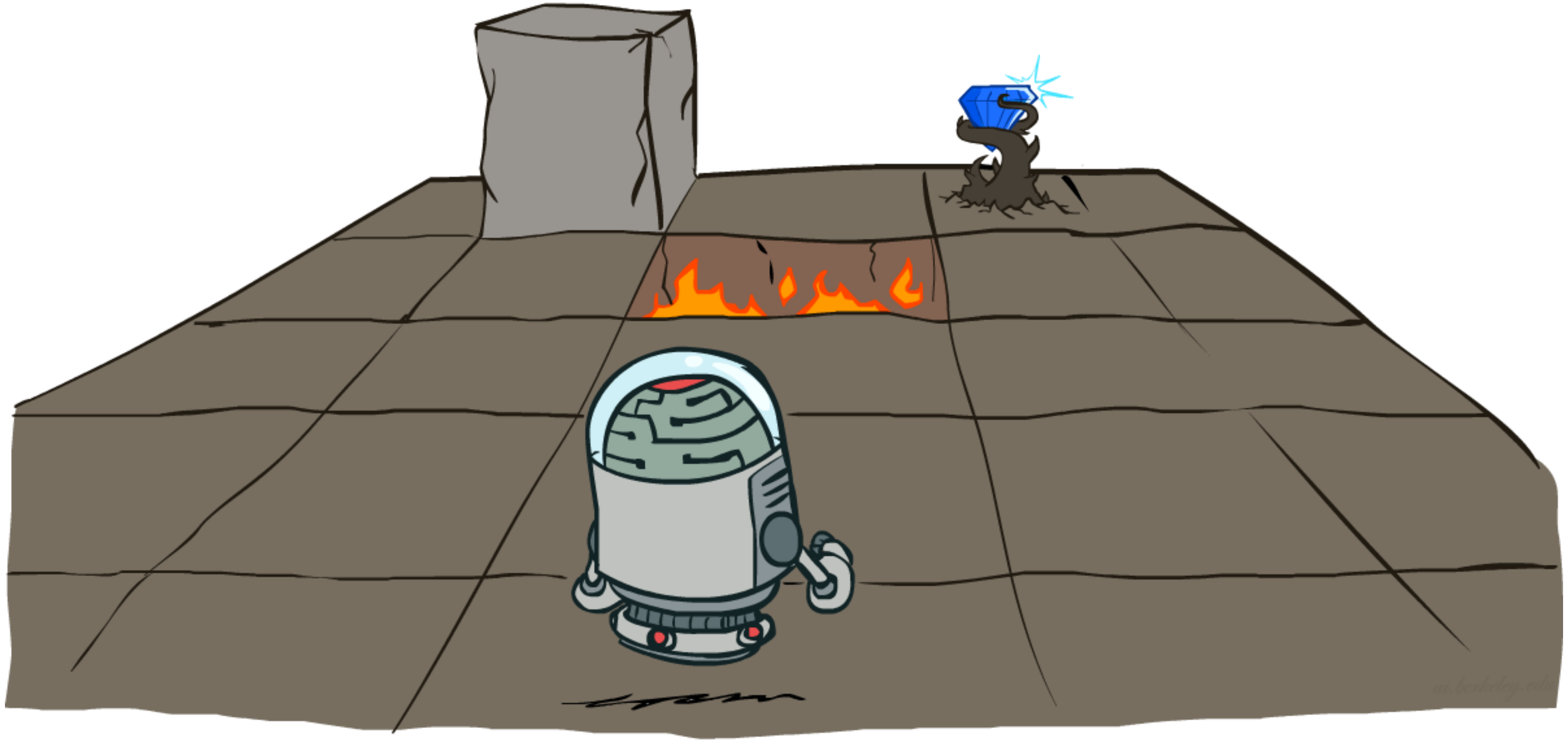
COE 4213564
Introduction to Artificial Intelligence
Markov Decision Processes



Many slides are adapted from CS 188 (http://ai.berkeley.edu), CS 322, CIS 521, CS 221, CS182, CS4420.

Sequential Decision Problems

- Sequential decision problems are a fundamental concept in computer science, operations research, and AI. They deal with making a series of decisions over time, where each choice affects future options and outcomes.
- Core Idea
  - A sequential decision problem involves multiple stages.
  - At each stage, an agent chooses an action.
  - The state of the system evolves based on the action taken.
  - The goal is to maximize rewards (or minimize costs) over the entire sequence.

Non-Deterministic Search

- A non-deterministic search problem is one where actions or algorithmic steps can lead to multiple possible outcomes, rather than a single predictable result.
- In computer science, a non-deterministic algorithm may produce different outputs for the same input on different runs.
- A non-deterministic search problem is a search task where the algorithm explores possible solutions by guessing and then verifying rather than deterministically following a fixed path.

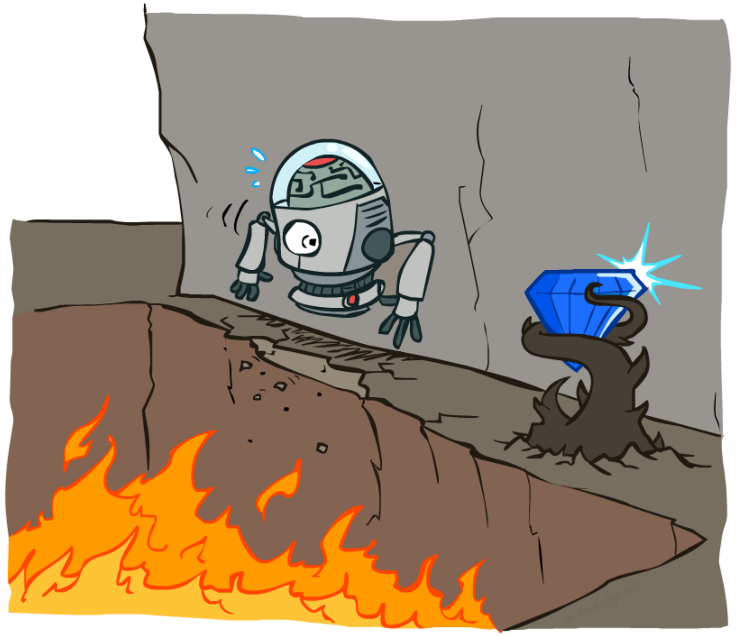
Stochastic Decision-Making

- Stochastic environment
  - A stochastic environment is one where the outcome of actions is uncertain and governed by probability distributions rather than deterministic rules. In other words, the same action taken in the same state can lead to different results at different times.
- Stochastic Decision-Making
  - Stochastic decision-making refers to making choices in environments where outcomes are uncertain and governed by probability distributions. Unlike deterministic settings, where actions lead to predictable results, stochastic environments require strategies that account for randomness and risk.
  - Core Characteristics:
    - Uncertainty: Actions don’t guarantee a single outcome; instead, they produce a range of possible results.
    - Probabilistic Transitions: Future states depend on probabilities, not certainties.
    - Sequential Nature: Decisions are often made step by step, with each choice influencing future options.
    - Optimization Goal: **Maximize expected reward** (or minimize expected cost) over time.

Markov Decision Processes

- Markov Decision Processes (MDPs) are indeed a type of non-deterministic search problem, because actions don't always lead to a single predictable outcome.
- A Markov Decision Process (MDP) is a mathematical framework for modeling sequential decision problems where:
  - The environment is fully observable (the agent knows the current state).
  - The environment is stochastic (actions lead to probabilistic outcomes).
  - The dynamics follow the Markov property (future depends only on the current state and action, not the full history).
  - Rewards are additive over time (the agent seeks to maximize cumulative reward).

Markov Decision Processes

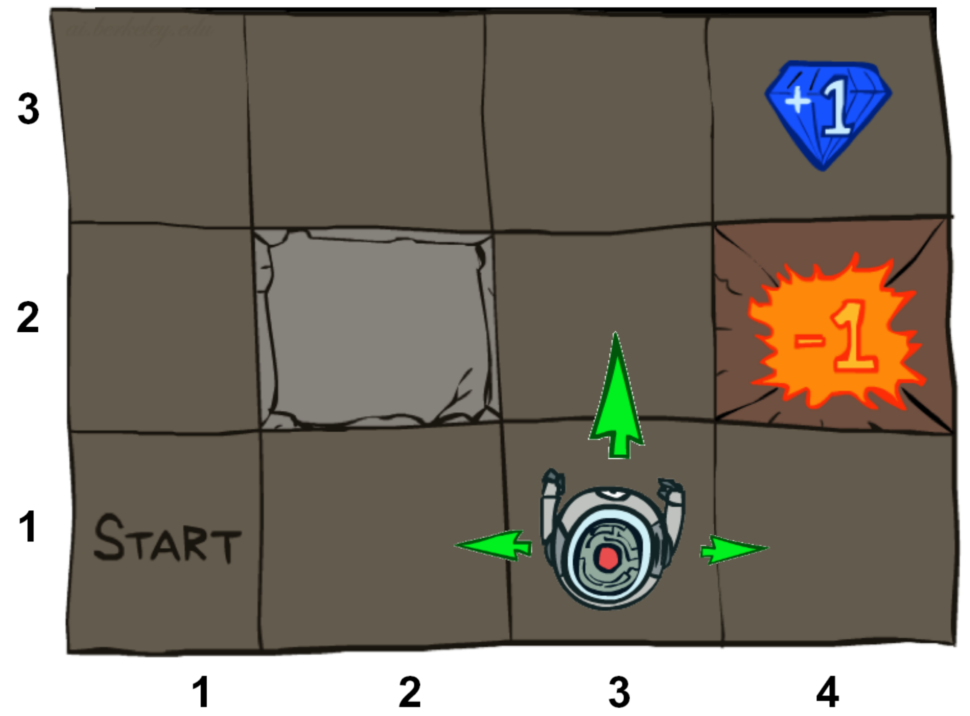


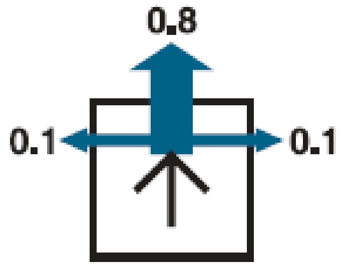
- MDPs are non-deterministic search problems.
- Markov decision Process (MDP) consists of
  - a set of states (with an initial state s_0);
  - a set ACTIONS(s) of actions in each state;
  - a transition model P(s' | s;a); and
  - a reward function R(s,a, s').
- Transitions are Markovian that means the probability of reaching state s' from s depends only on s and not on the history of earlier states.

Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or –1.
- The actions available to the agent in each state are given by ACTIONS(s), sometimes abbreviated to A(s).
- In the 4x3 environment, the actions in every state are Up (North), Down (South), Left (West), and Right (East).
- If the environment were deterministic, a solution would be easy:

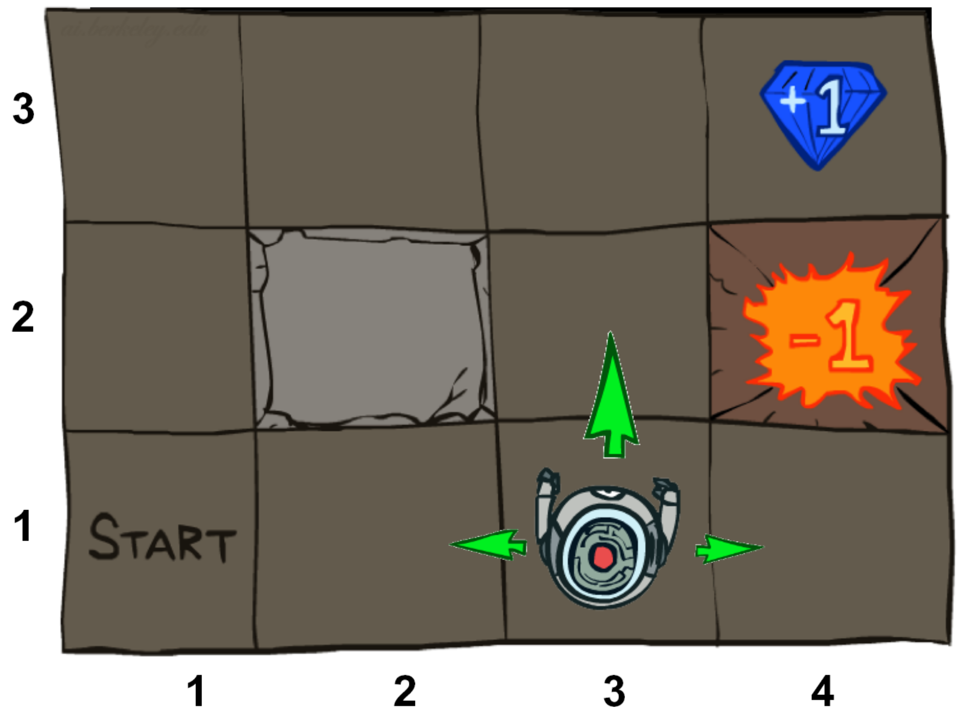
  [Up, Up, Right, Right, Right].
- Unfortunately, the environment won't always go along with this solution, because the actions are unreliable (non-deterministic)

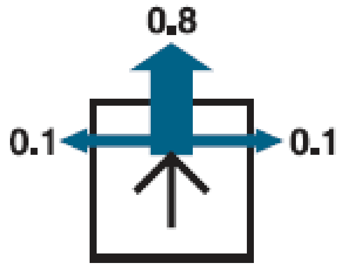




Example: Grid World

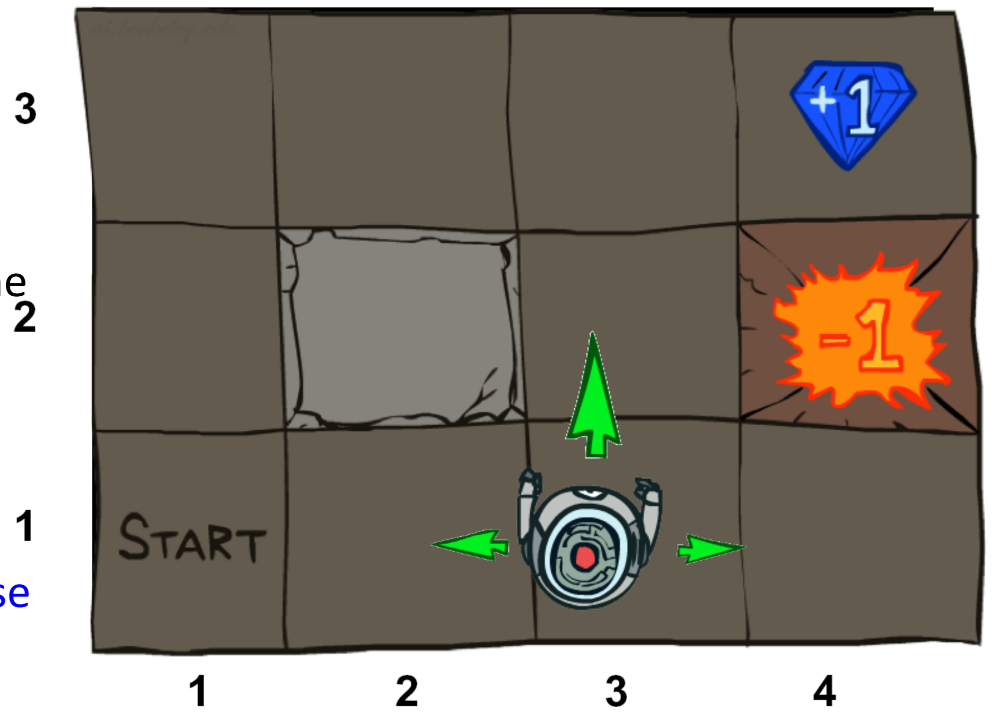
- We are in a non-deterministic, stochastic or noisy environment.
- Noisy movement: actions do not always go as planned
  - With 80% of the time, the action Up (North) takes place (if there is no wall there)
  - With 10% of the time, the agent moves Left (West) or Right (East)
  - If there is a wall in the direction the agent would have been taken, the agent stays there
- The transition model (or just "model," when the meaning is clear) describes the outcome of each action in each state.
- Here, the outcome is stochastic, so we write transition functions P(s' | s, a) (or T(s ,a, s')) for the probability of reaching state s' if action a is done in state s.
- We will assume that transitions are Markovian: the probability of reaching s' from s depends only on s and not on the history of earlier states.

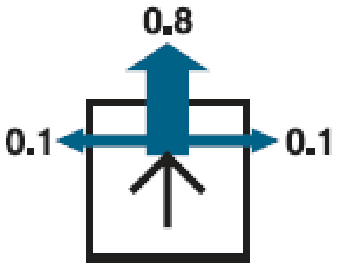




Example: Grid World

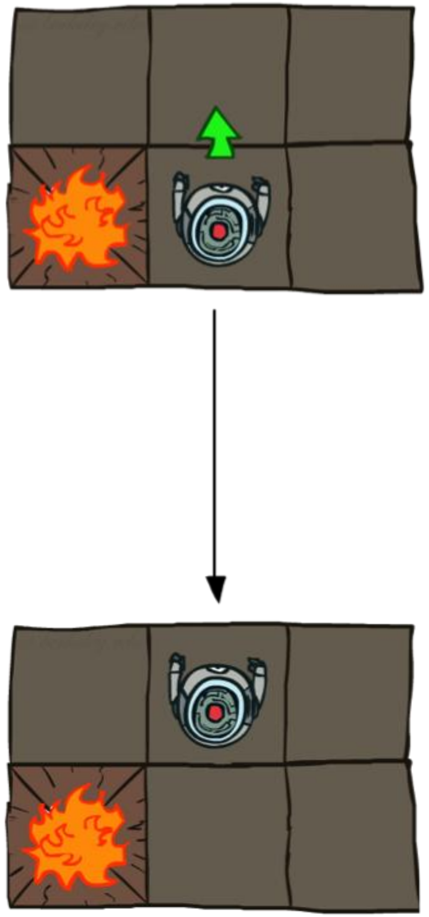
- The agent receives rewards each time step
  - Small “living” reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- To complete the definition of the task environment, we must specify the utility function for the agent. Because the decision problem is sequential, the utility (reward) function will depend on a sequence of states and actions.
- For every transition from s to s’ via action a, the agent receives a reward Reward R(s, a, s’). The rewards may be positive or negative, but they are bounded by -Rmax and +Rmax.
- For our particular example, the reward is -0.04 for all transitions except those entering terminal states (which have rewards +1 and –1). The utility of an environment history is just (for now) the sum of the rewards received.
- For example, if the agent reaches the +1 state after 10 steps, its total utility will be (9 x -0.04)+1=0.64. The negative reward of –0.04 gives the agent an incentive to reach (4,3) quickly, so our environment is a stochastic generalization of the search problems of Chapter 3.
- Goal: maximize sum of rewards



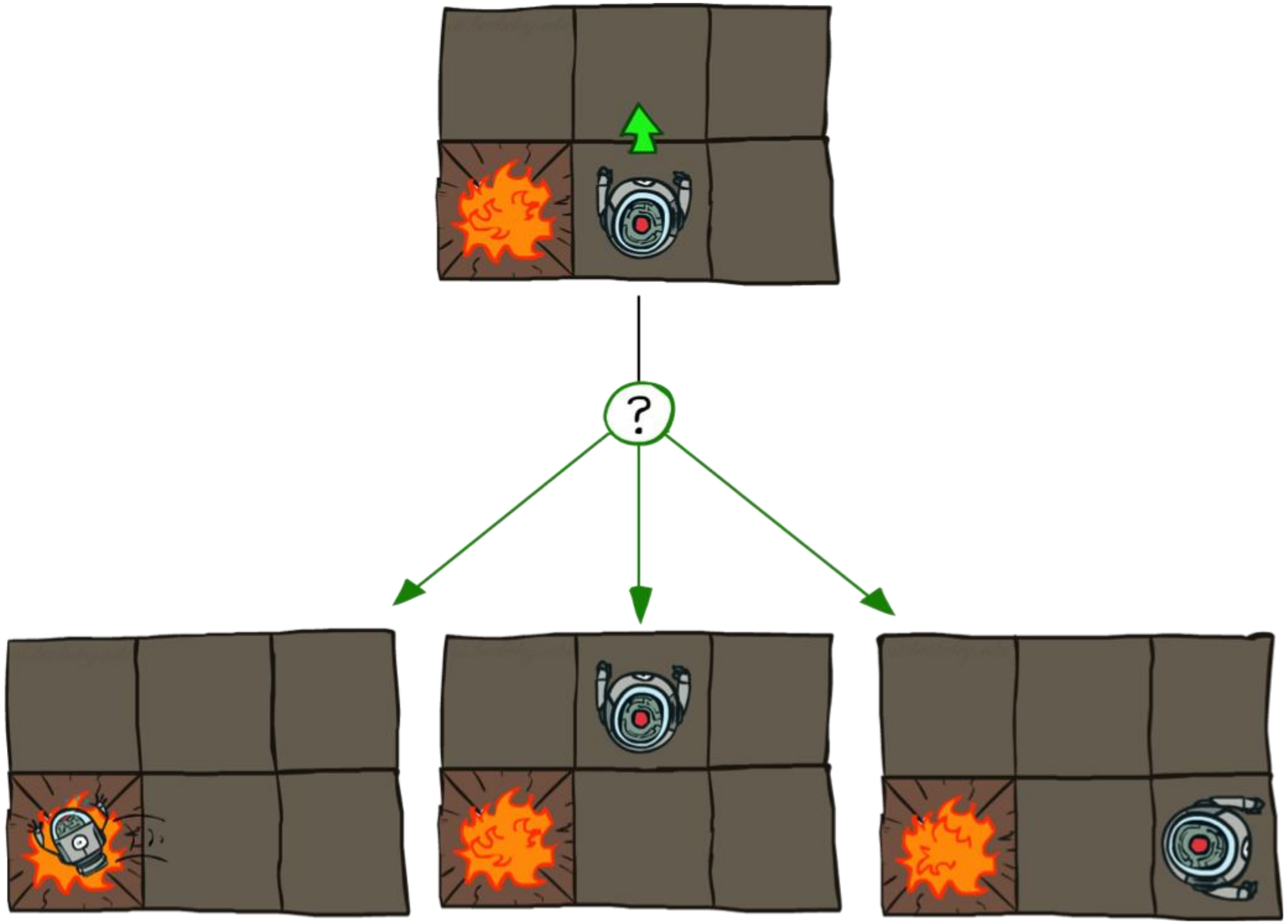


Grid World Actions

Deterministic Grid World

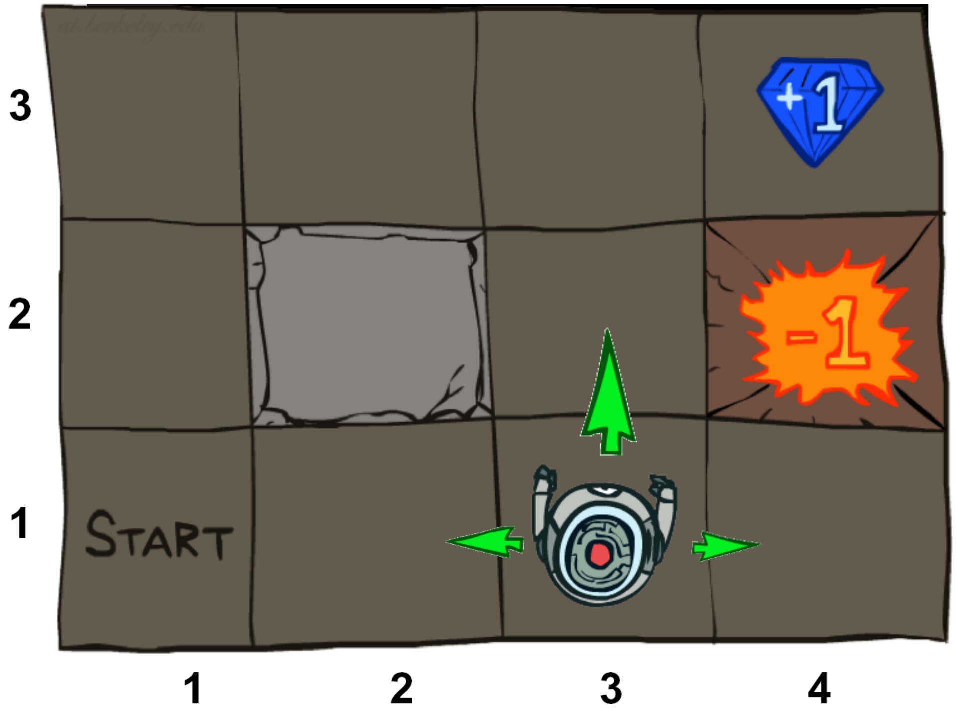


Stochastic Grid World



Markov Decision Processes

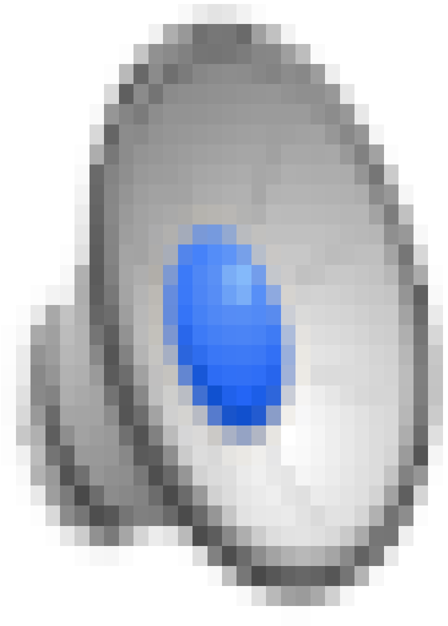
- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s' | s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
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

[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

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

$$=$$

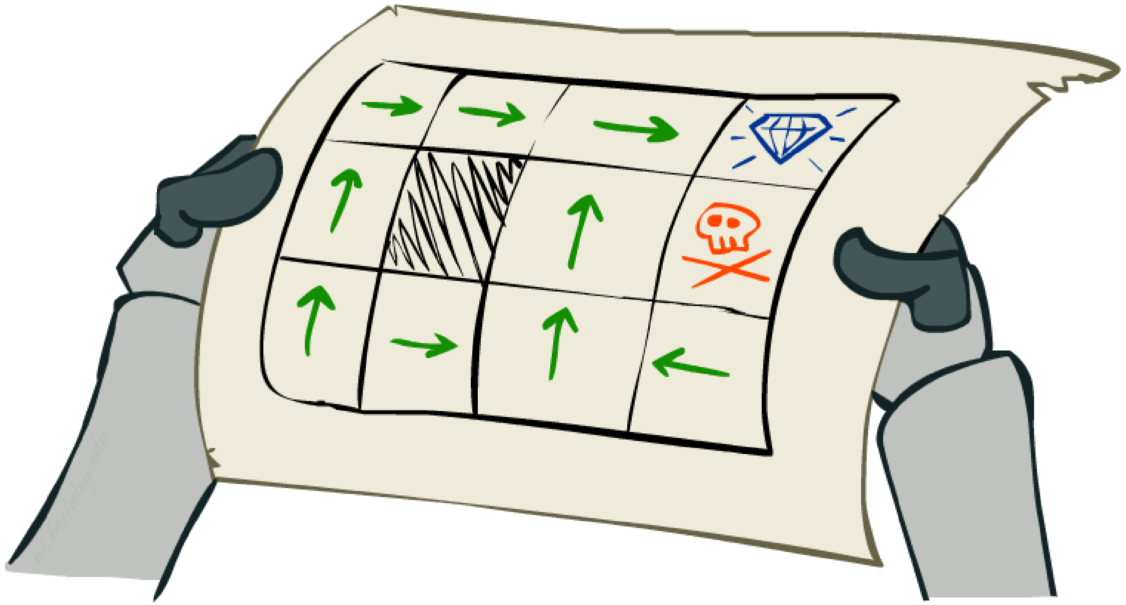
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

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

Andrey Markov
(1856-1922)

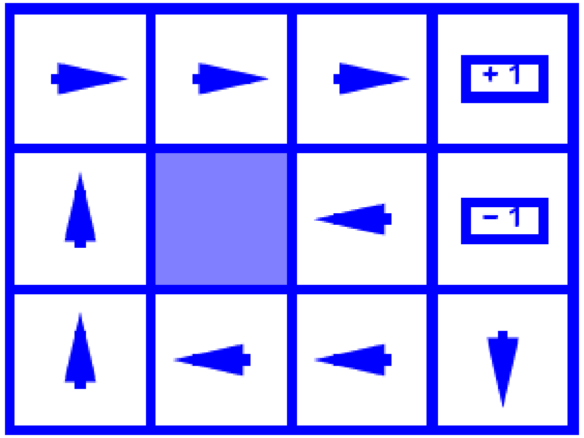
Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal as a solution.
- In MDPs, a solution is called a policy that will take the agent from start state to goal state.
- It is traditional to denote a policy by π,
- and π (s) is the action recommended by the policy π for state s.
- 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 compute entire policies
  - It computed the action for a single state only

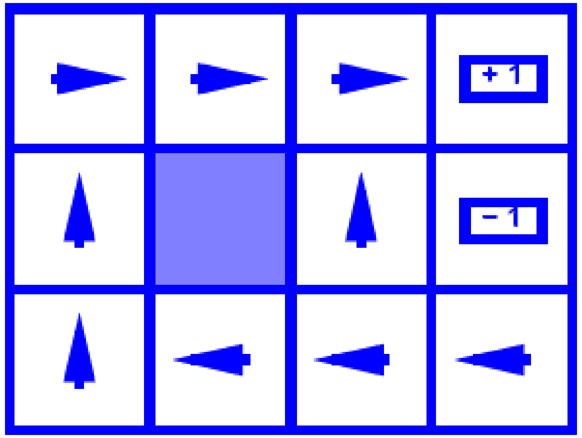


Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

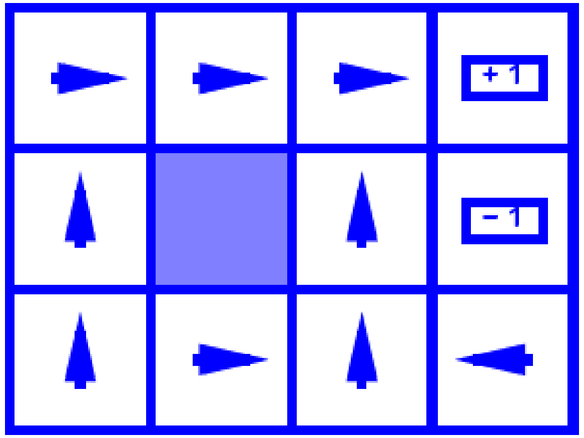
Optimal Policies



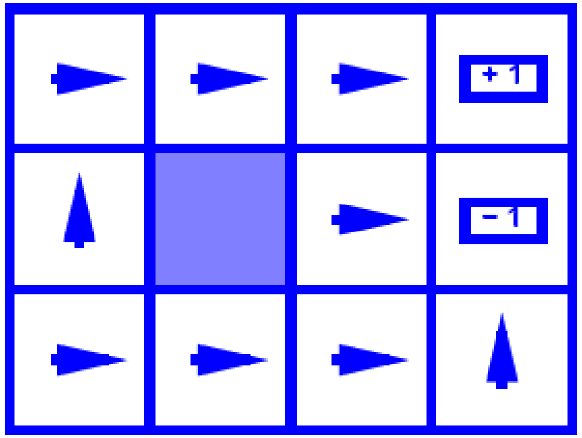
R(s) = -0.01



R(s) = -0.03

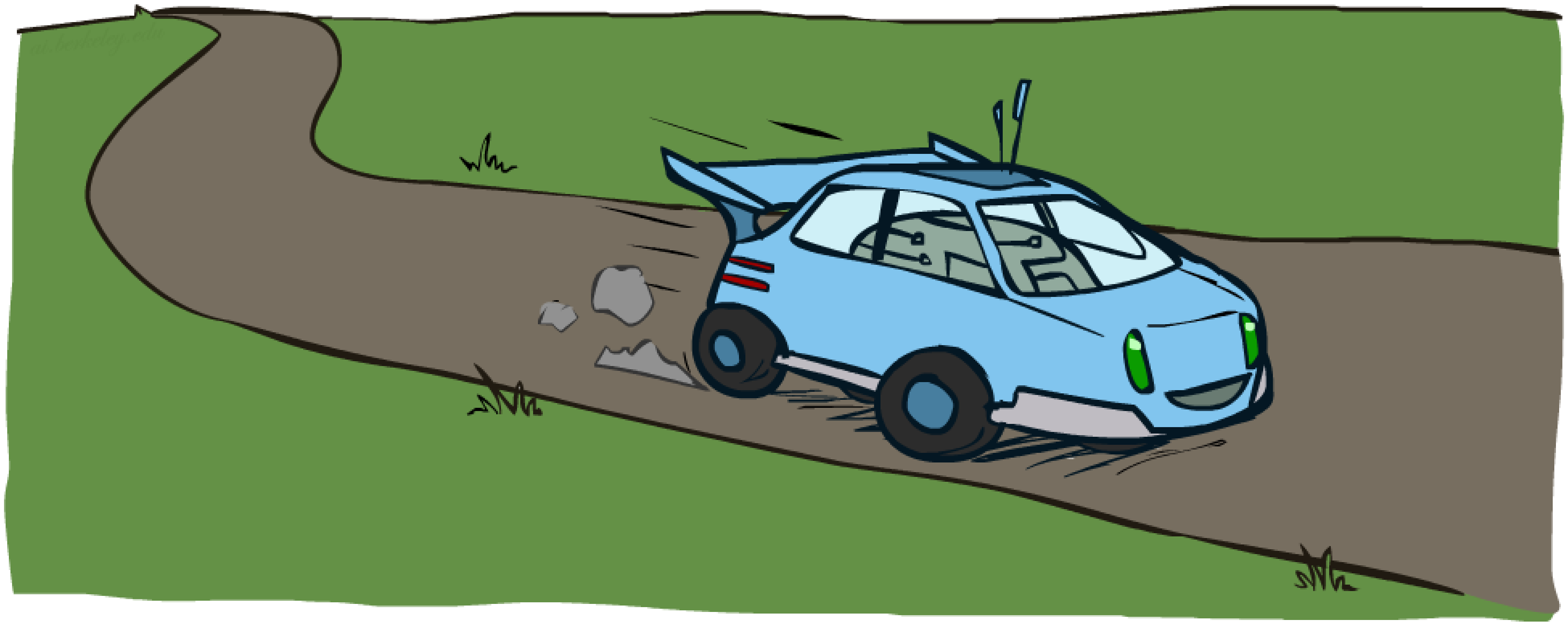


R(s) = -0.4



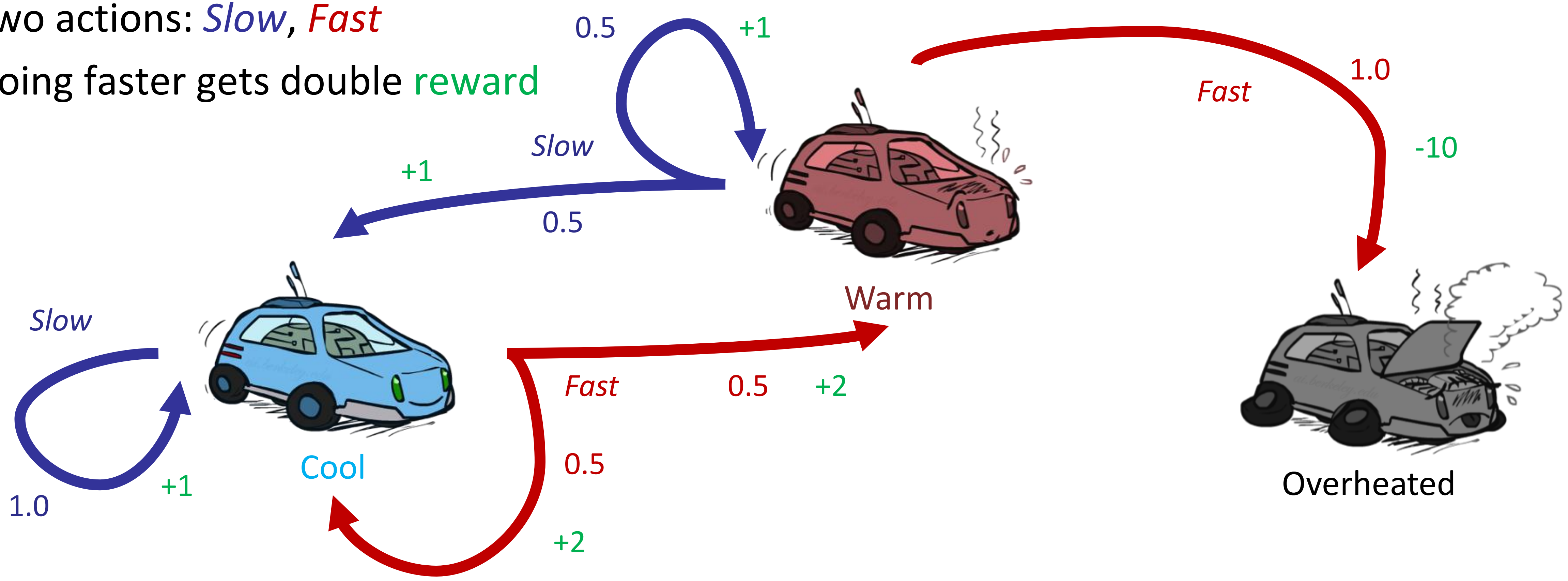
R(s) = -2.0

Example: Racing

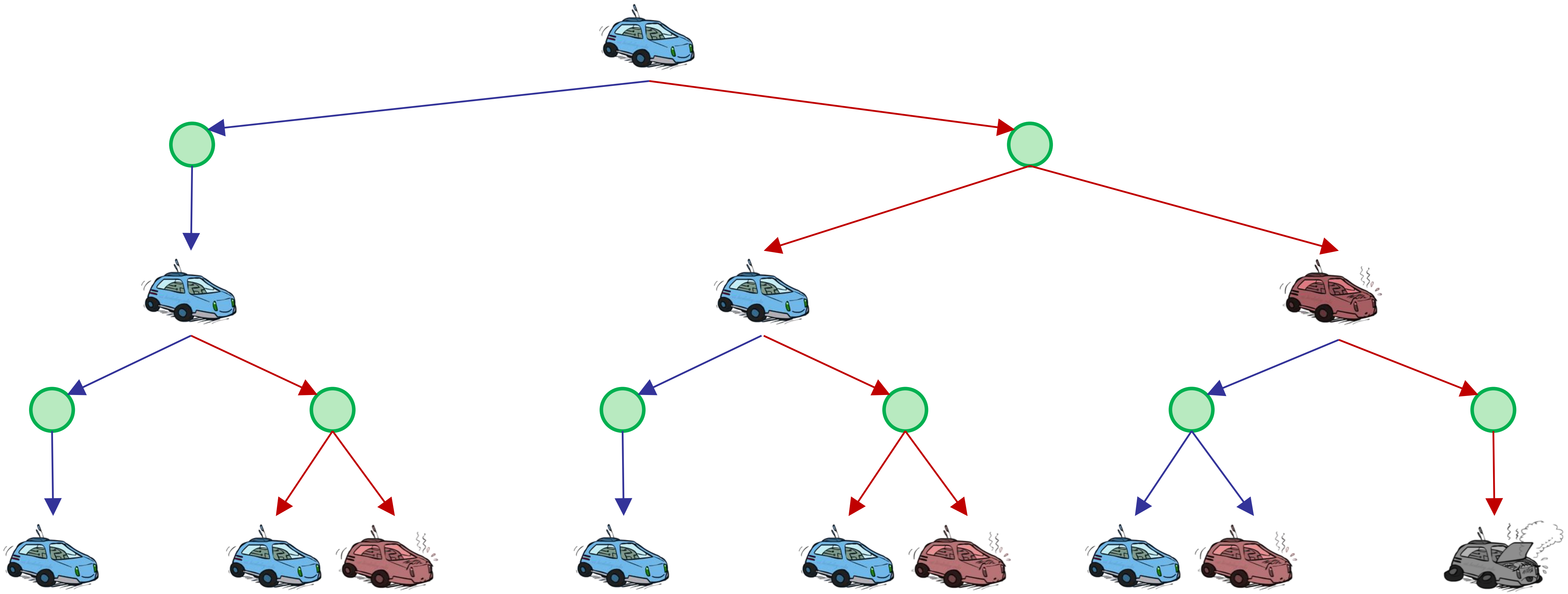


Example: Racing

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

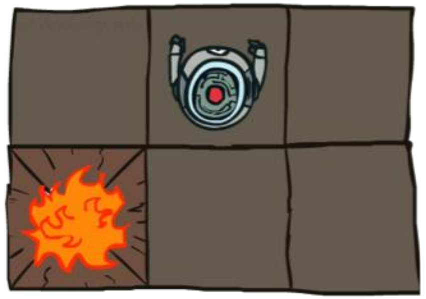


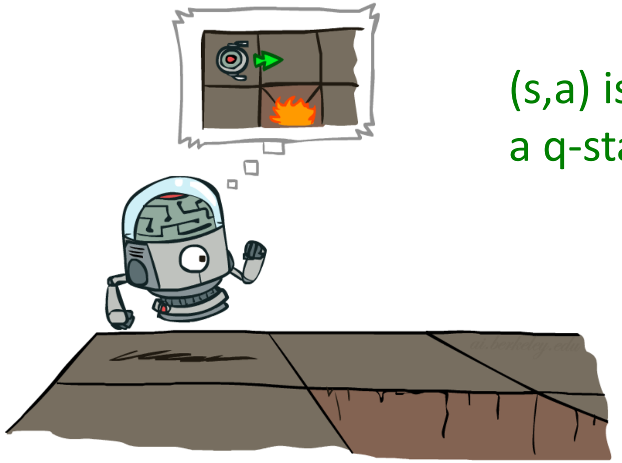
Racing Search Tree

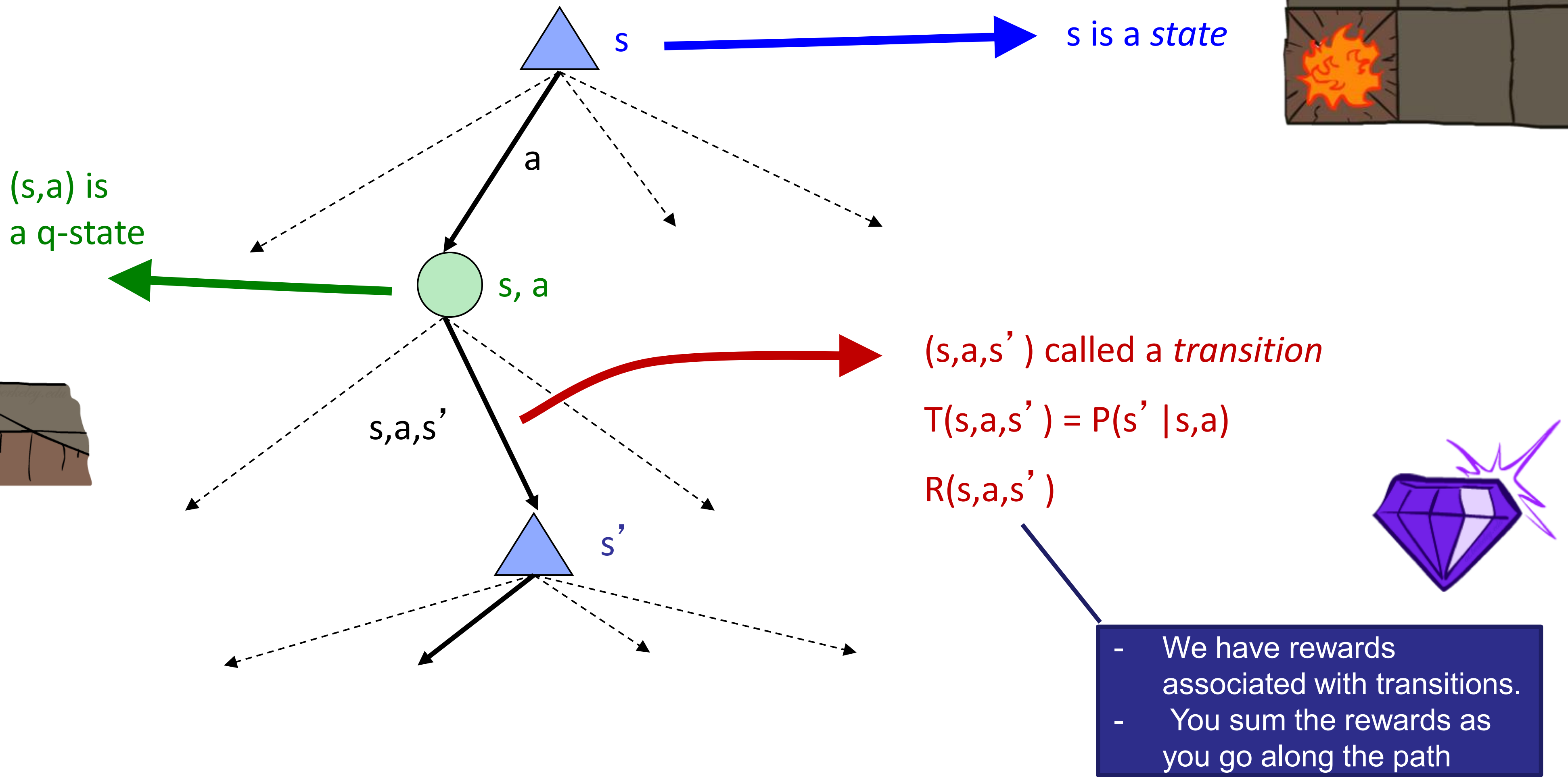


MDP Search Trees

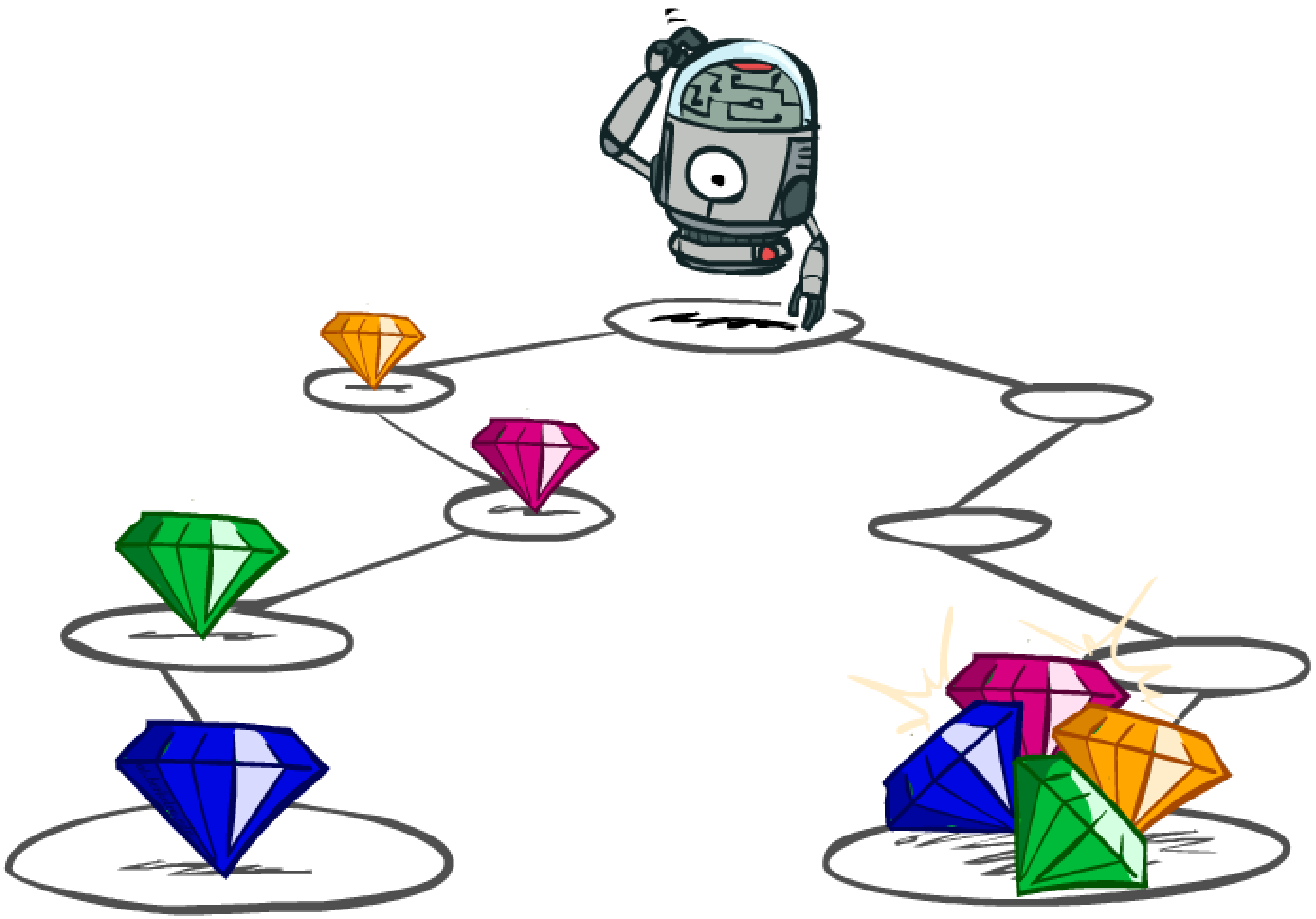
- Each MDP state projects an expectimax-like search tree





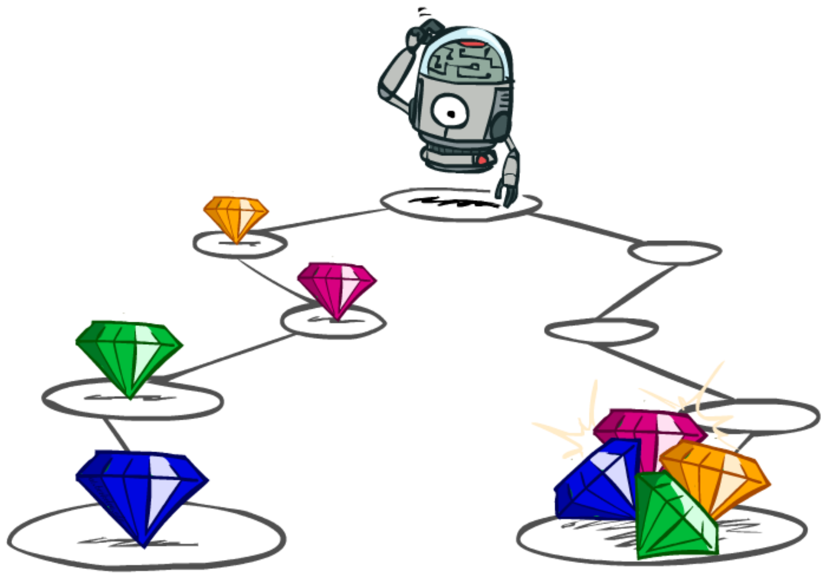


Utilities of Sequences



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]
- Sooner is better



Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later. We use a discount factor γ which is a number between 0 and 1.
- One solution: values of rewards decay exponentially

1

Worth Now

γ

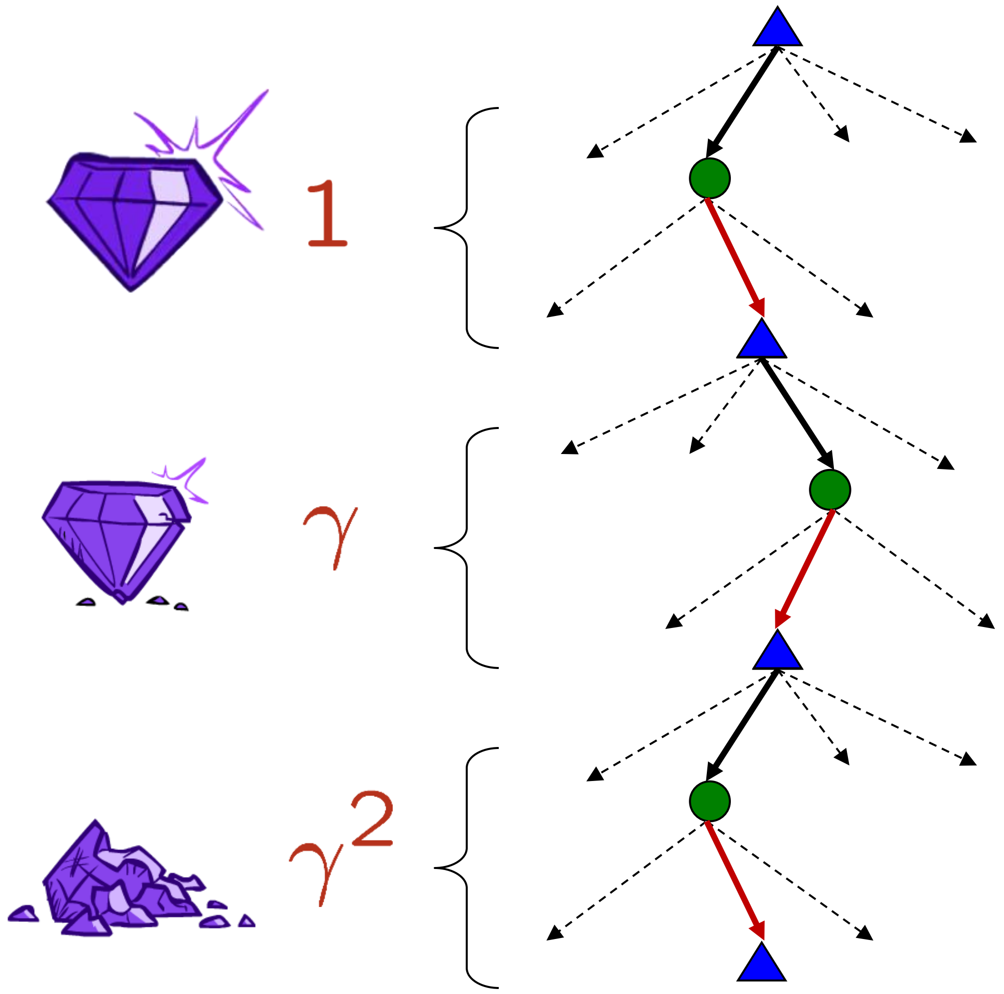
Worth Next Step

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
  - Reward 1 at 1st step, 2 at the 2nd, 3 at the 3rd steps
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])

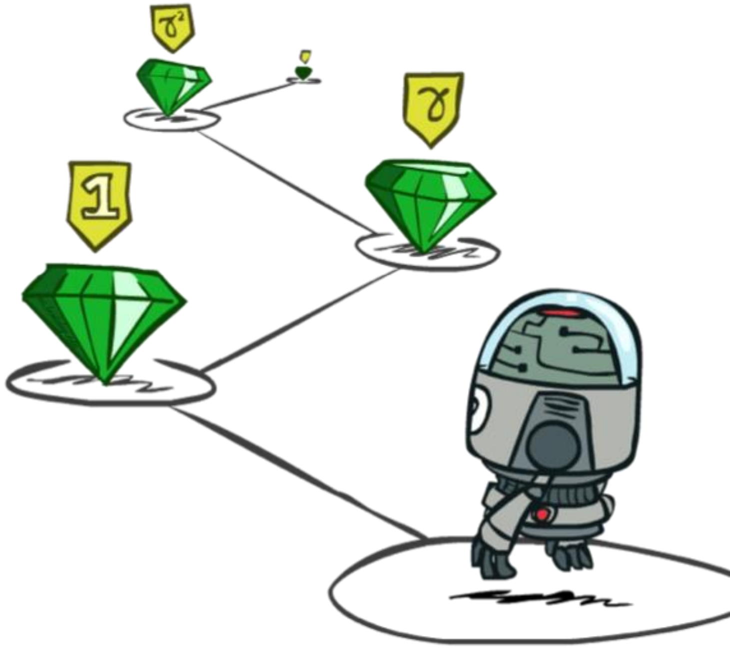


Stationary Preferences

- A policy that depends on the time is called nonstationary.
- An optimal action depends only on the current state, and then the optimal policy is stationary.
- Theorem: if we assume stationary preferences over a sequence of rewards:

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

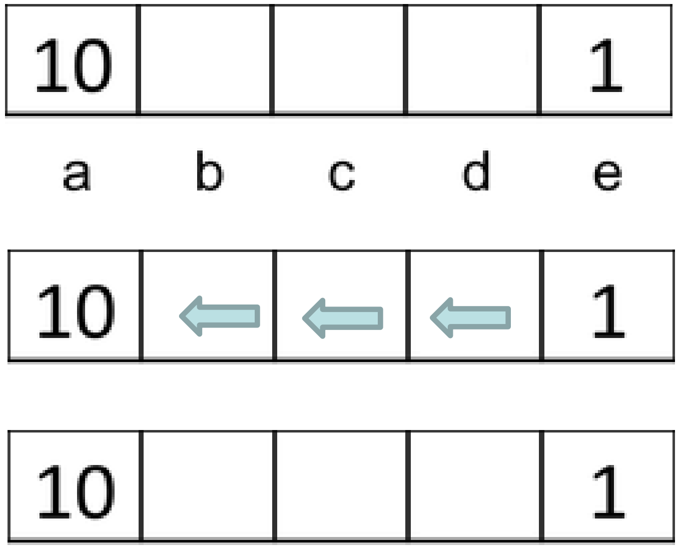
Reward sequences of a's are better sequences of b's. Add a new reward r to sequences



- Where r is the additional reward
- 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)
  - There is a reward of 10 at state a and 1 at state.
  - Transitions: deterministic



- Quiz 1: For $\gamma = 1$, what is the optimal policy?
- Quiz 2: For $\gamma = 0.1$, what is the optimal policy?

For Quiz 2 on state d:
- Sum rewards
- Go to east : $0 + \gamma * 1 = 0.1$ from d.
- Go to west : $0 + \gamma * 0 + \gamma^2 * 0 + \gamma^3 * 10 = 0.01$ from d.
- So it is better to go to east in you are in state d
- In other states b and c, go to west

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

$\gamma = 1 / \text{sqrt}(10)$

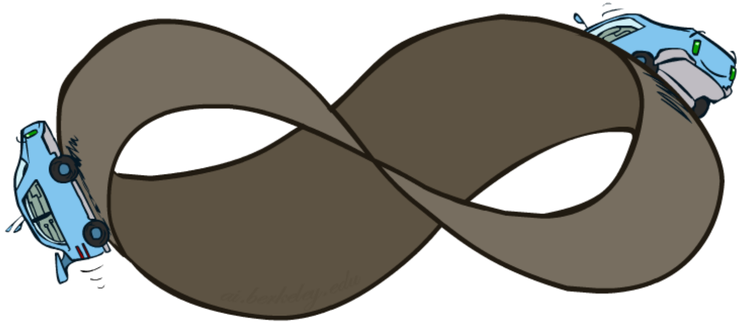
Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- if the environment does not contain a terminal state, or if the agent never reaches one, then all environment histories will be infinitely long, and utilities with additive undiscounted rewards will generally be infifinite

- 3 Solutions:
  - 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 $0 < \gamma < 1$

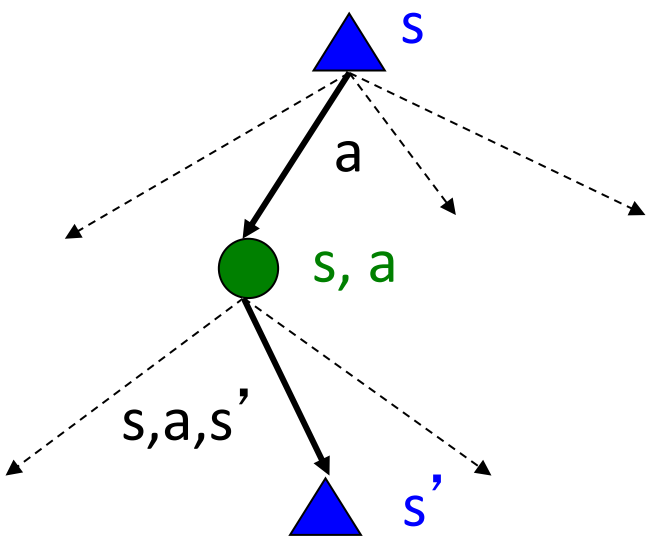
$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R\text{max}/(1-\gamma)$$

    - Sum of rewards are bounded (R_{max} : Maximum reward)
    - 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)



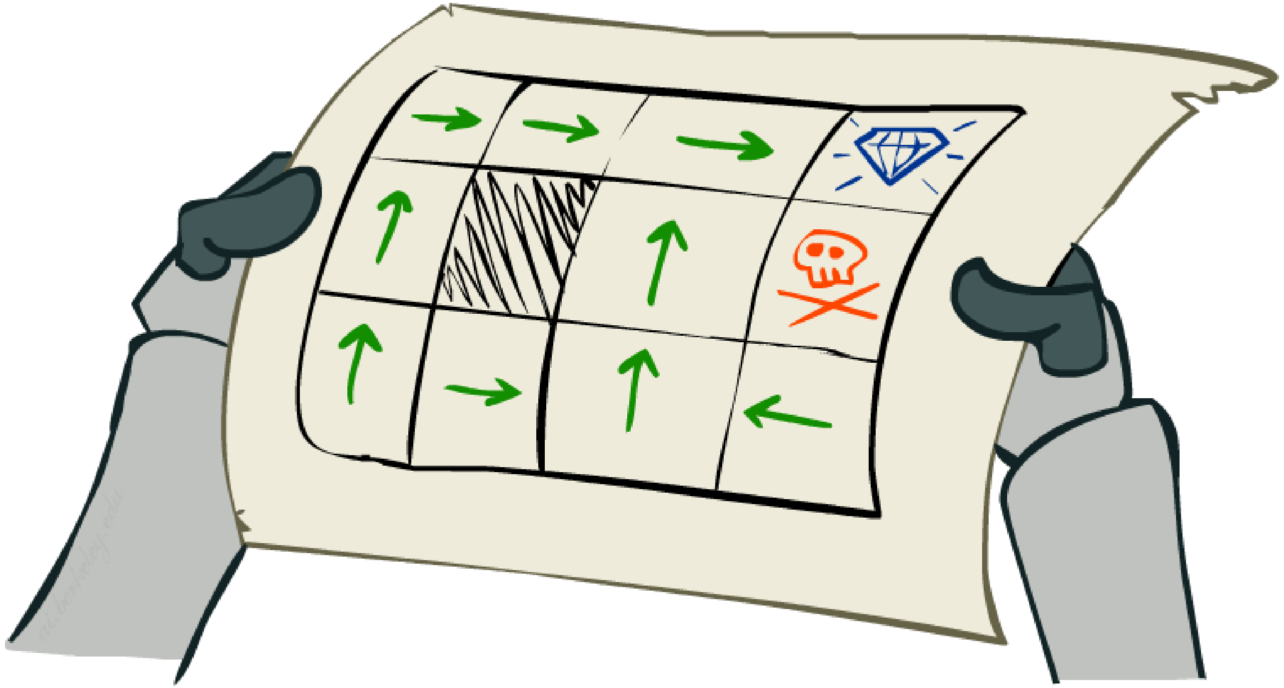
Recap: Defining MDPs

- 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



In this section, we present two different algorithms for solving MDPs:
- value iteration, and
- policy iteration.

Optimal Quantities

- The value (utility) of a state s:

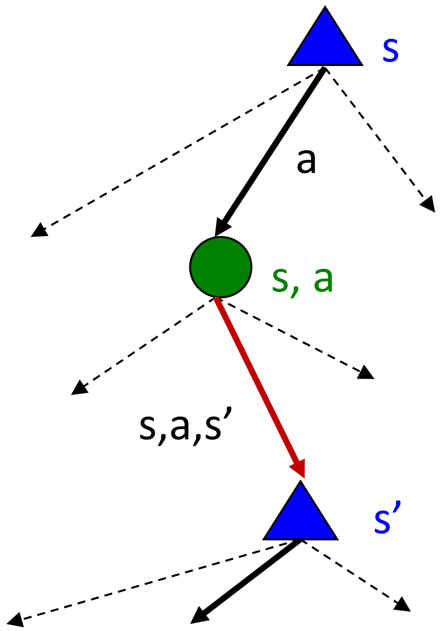
  $V^*(s)$ = expected utility starting in s and acting optimally

- 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



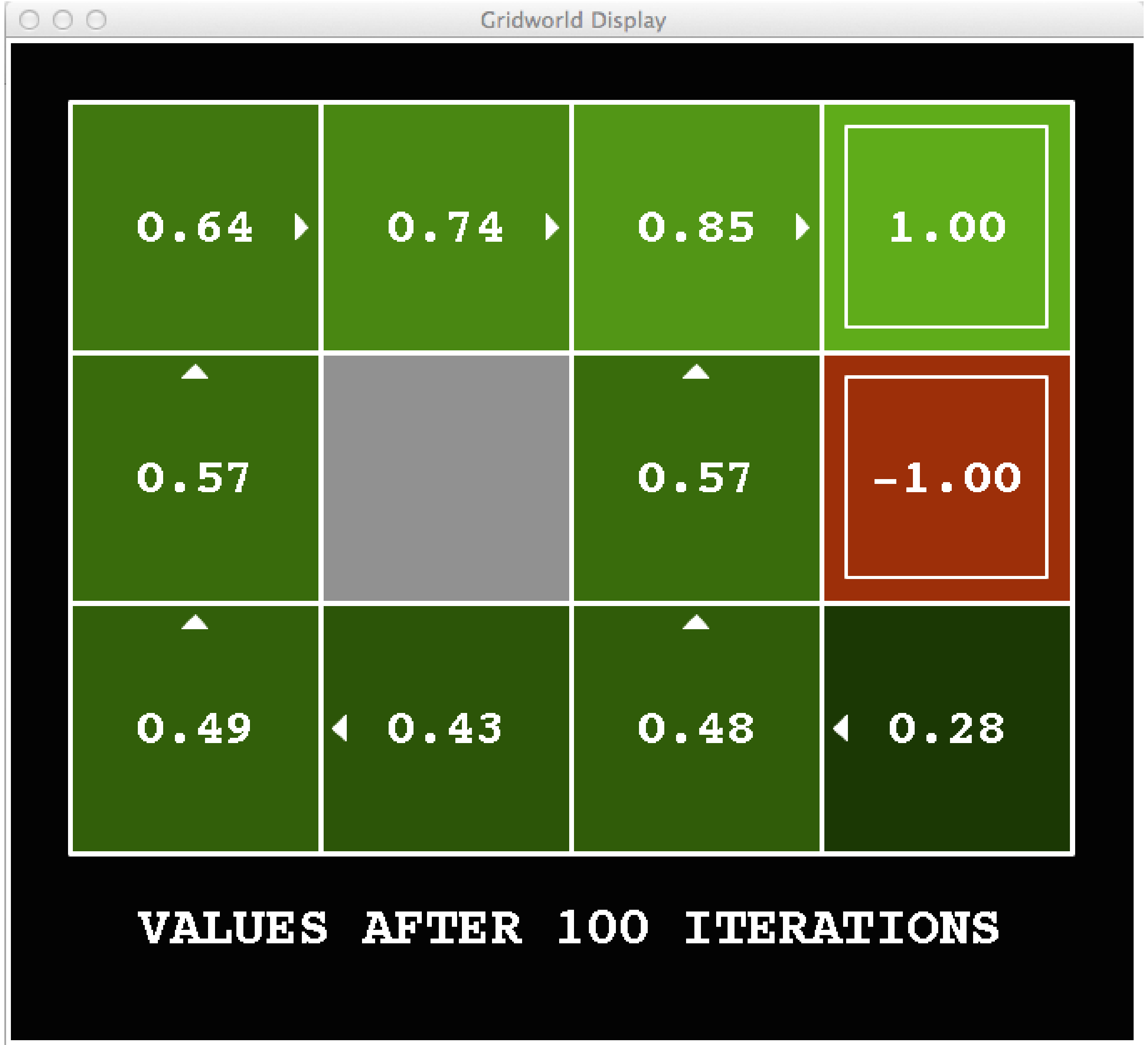
s is a *state*

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

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

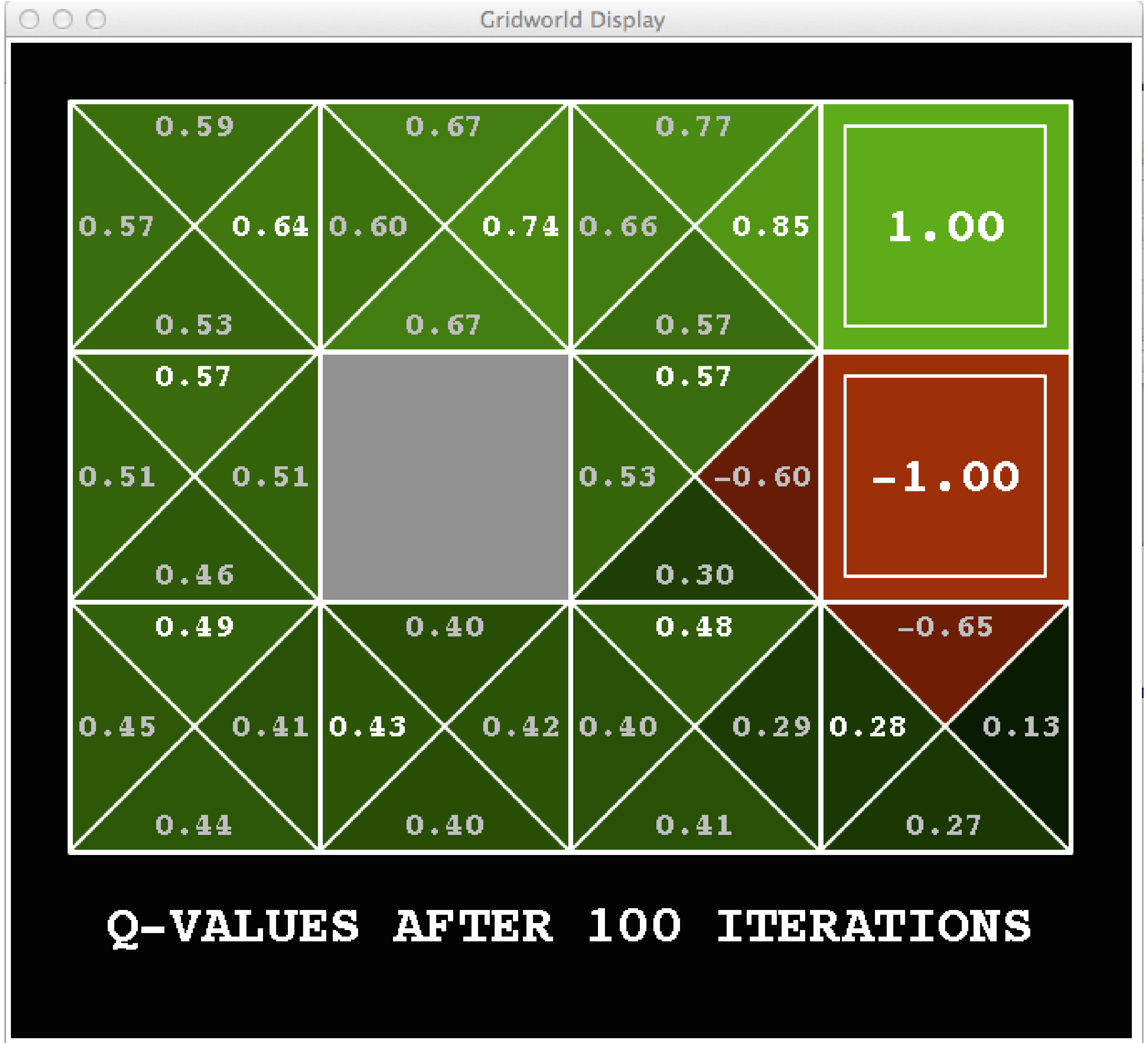
[Demo – gridworld values (L8D4)]

Snapshot of Demo – Gridworld V Values



Noise = 0.2
Discount = 0.9
Living reward = 0

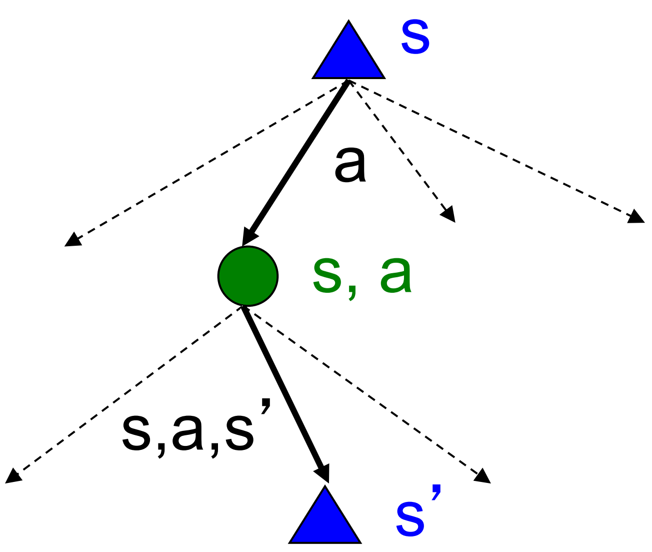
Snapshot of Demo – Gridworld Q Values



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:

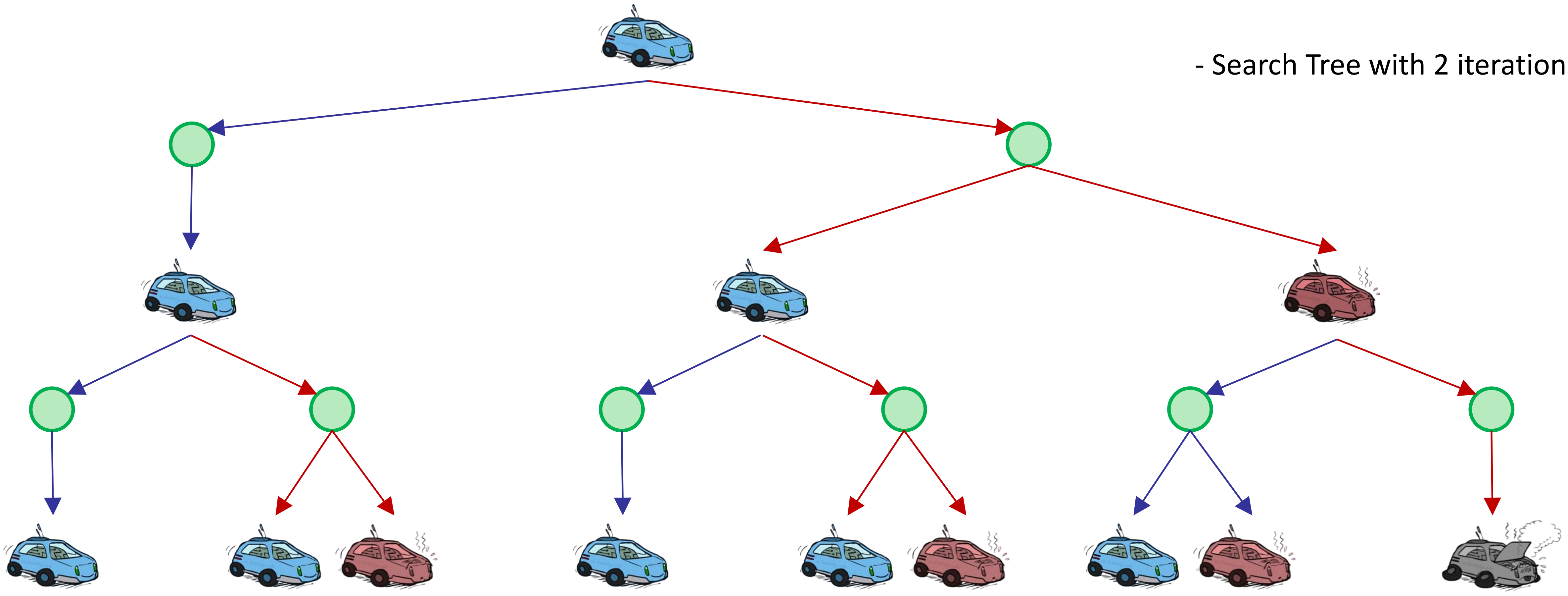
$$V^*(s) = \max_a Q^*(s,a)$$

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

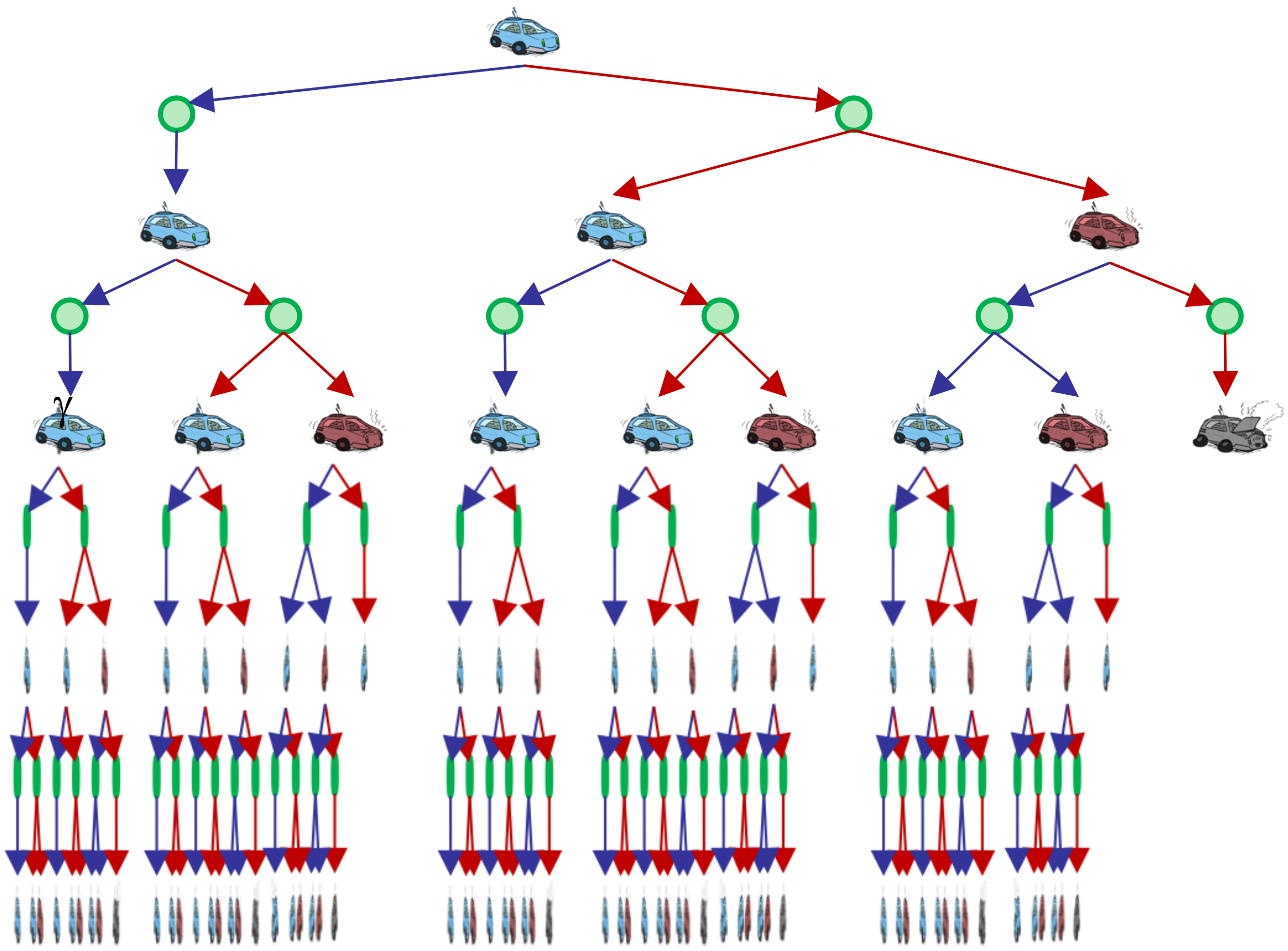
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

Racing Search Tree

- Search Tree with 2 iterations



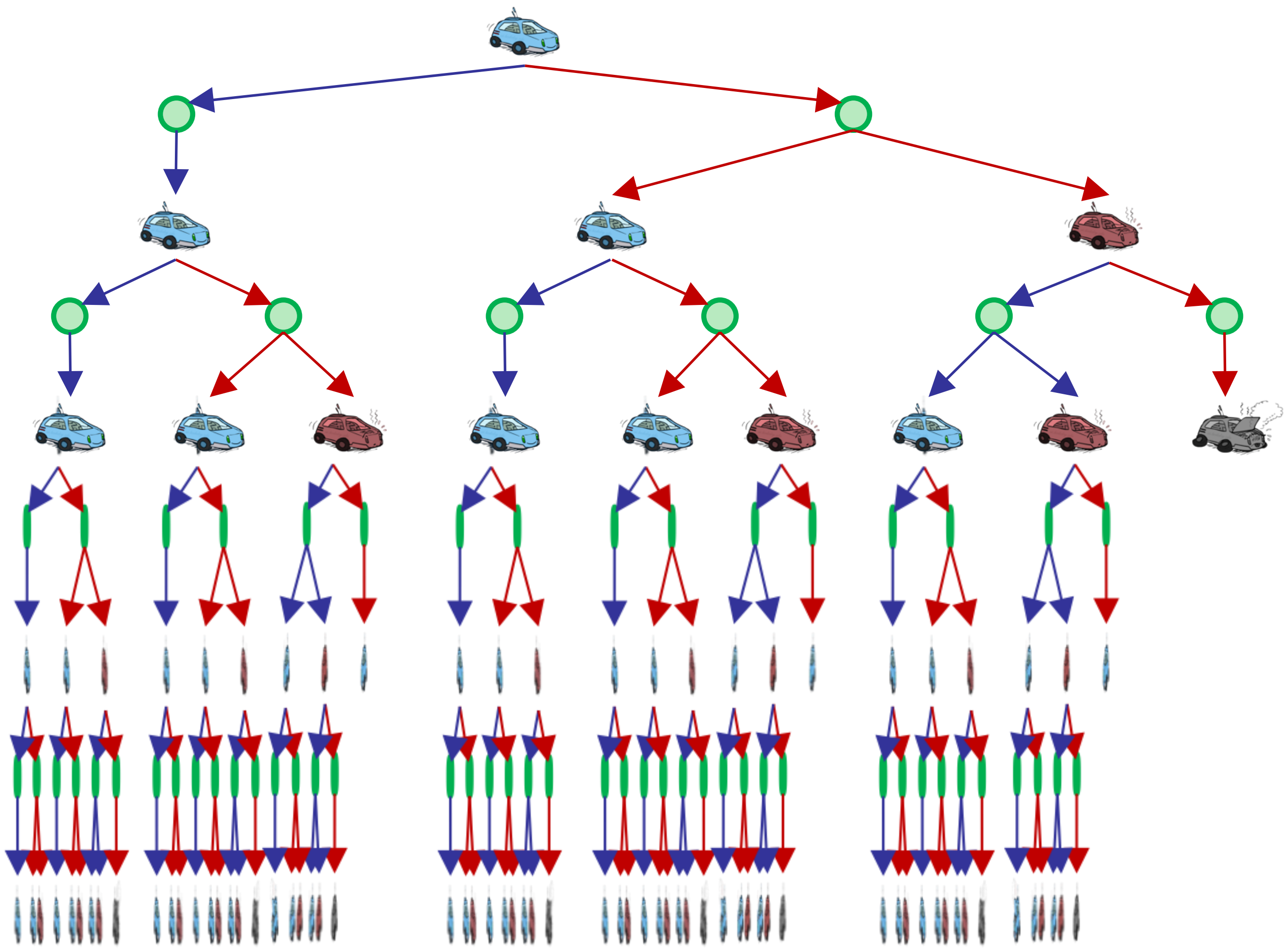
Racing Search Tree



- We play the game long time, not stop after 2 iterations
- Some branches are the same (repetitions)
- Use caching or bottom-up dynamic programming

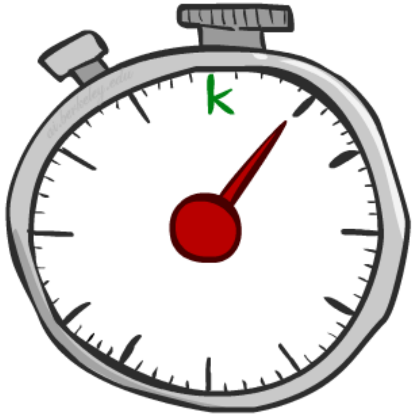
Racing Search Tree

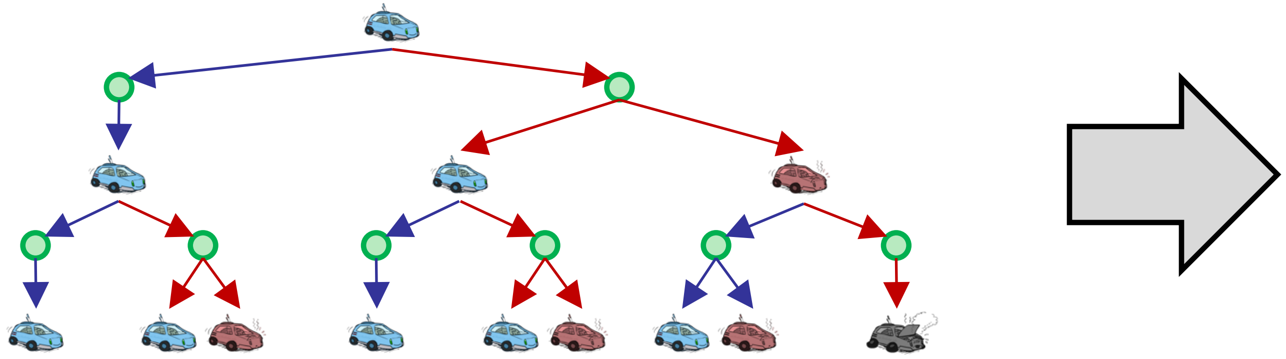
- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter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

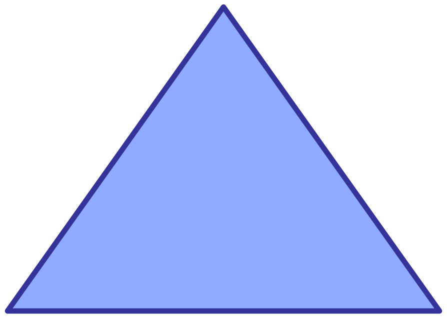


Time-Limited Values

- 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

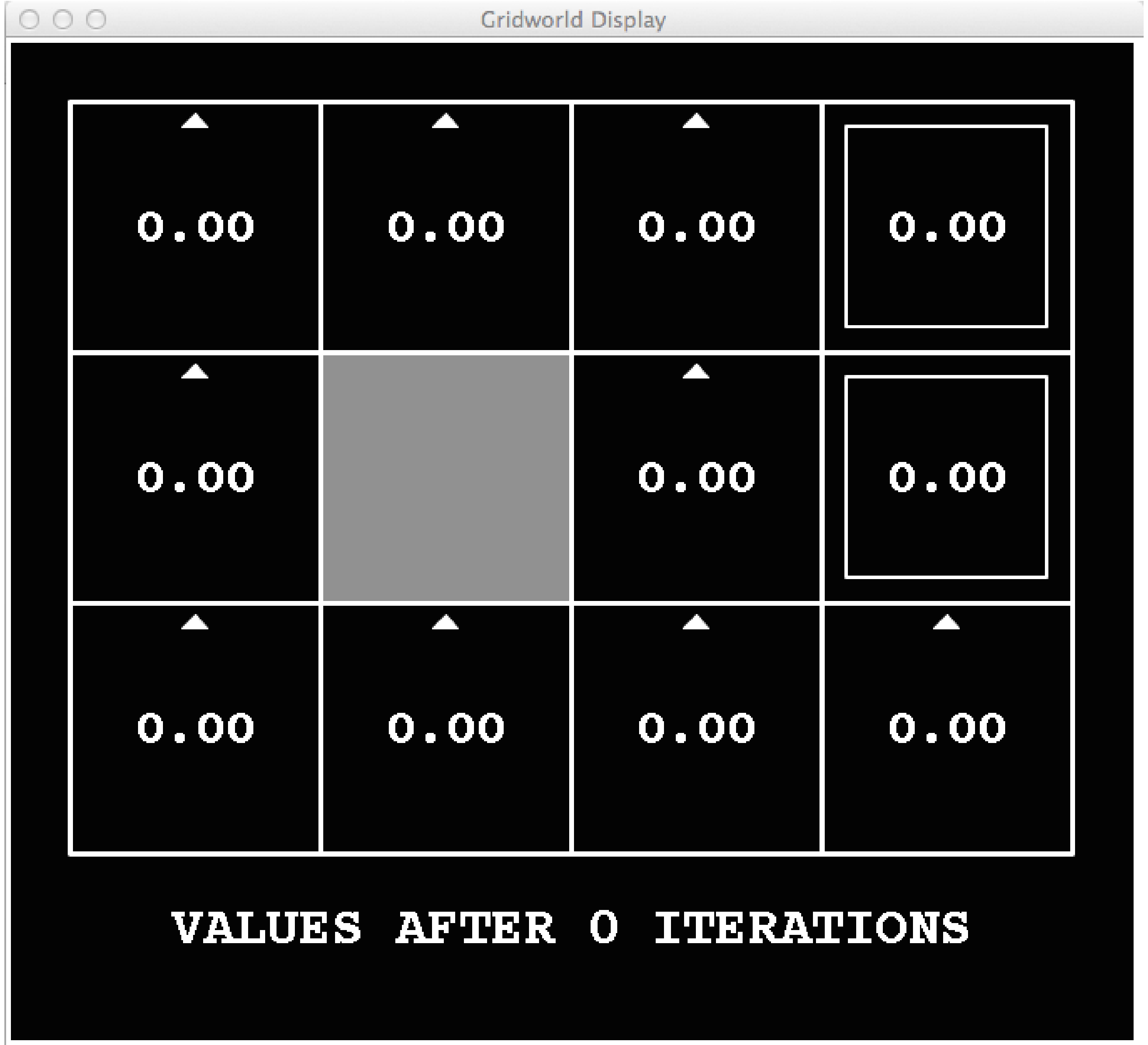




$V_2($  $)$

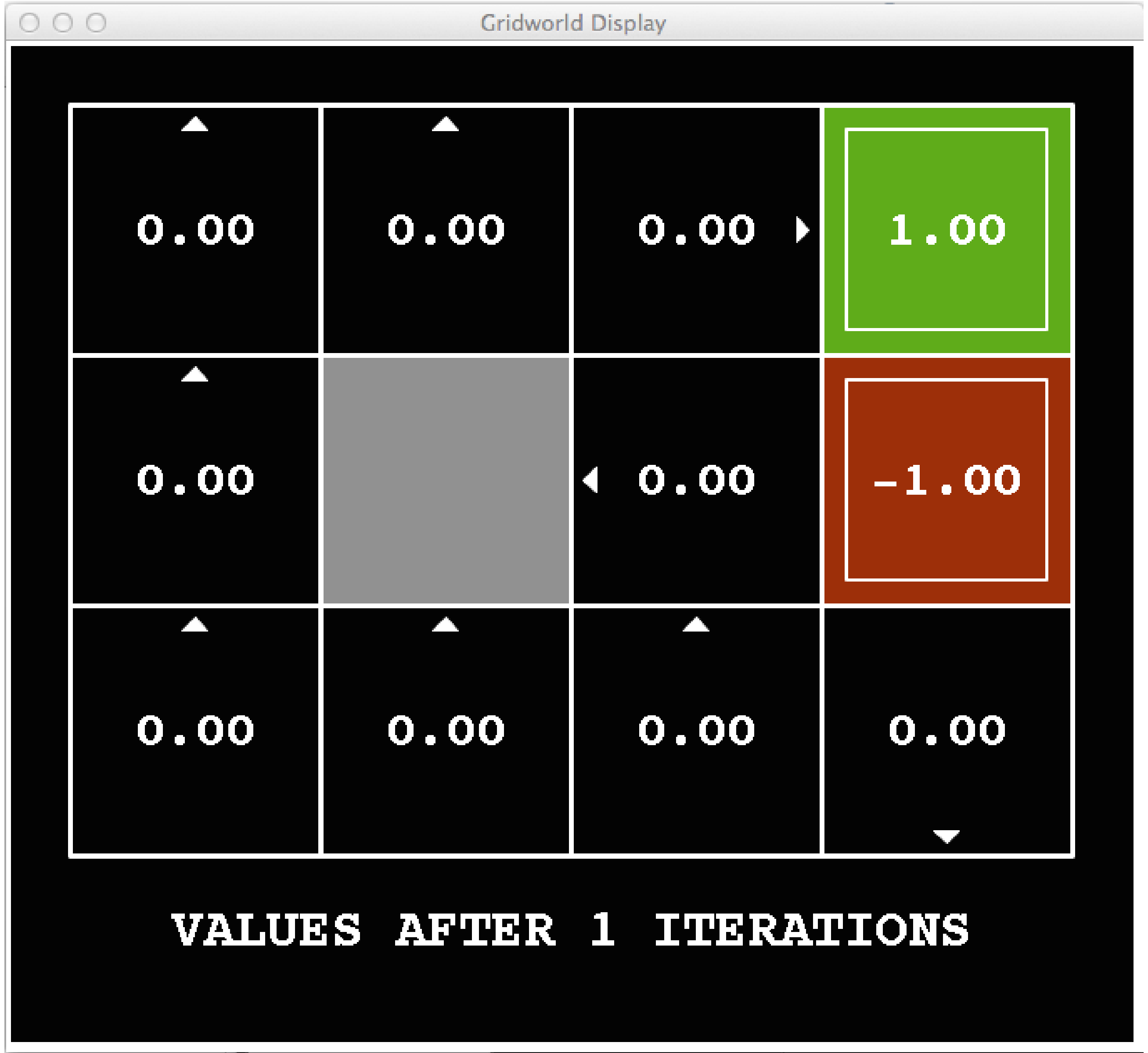
[Demo – time-limited values (L8D6)]

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

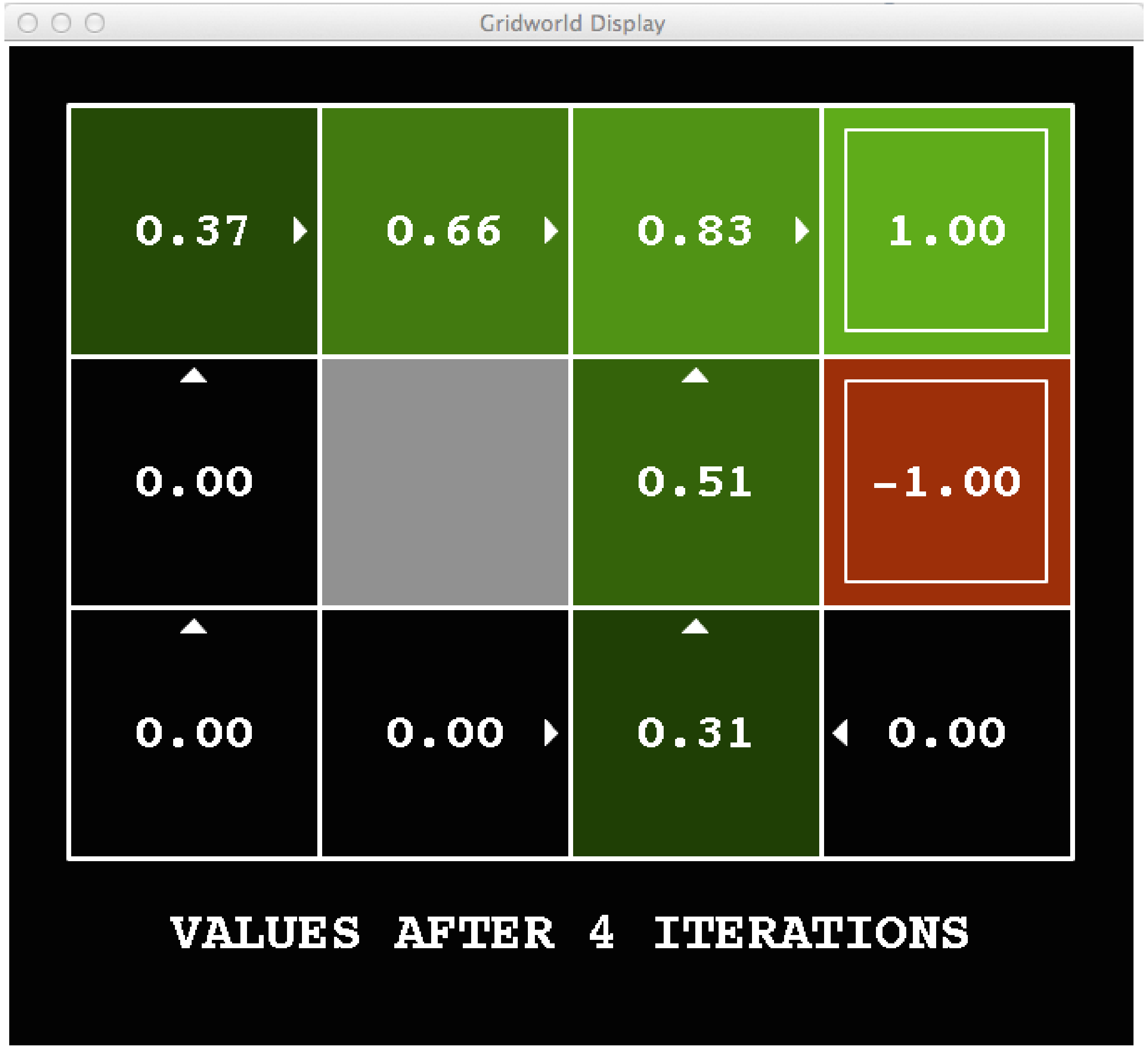
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

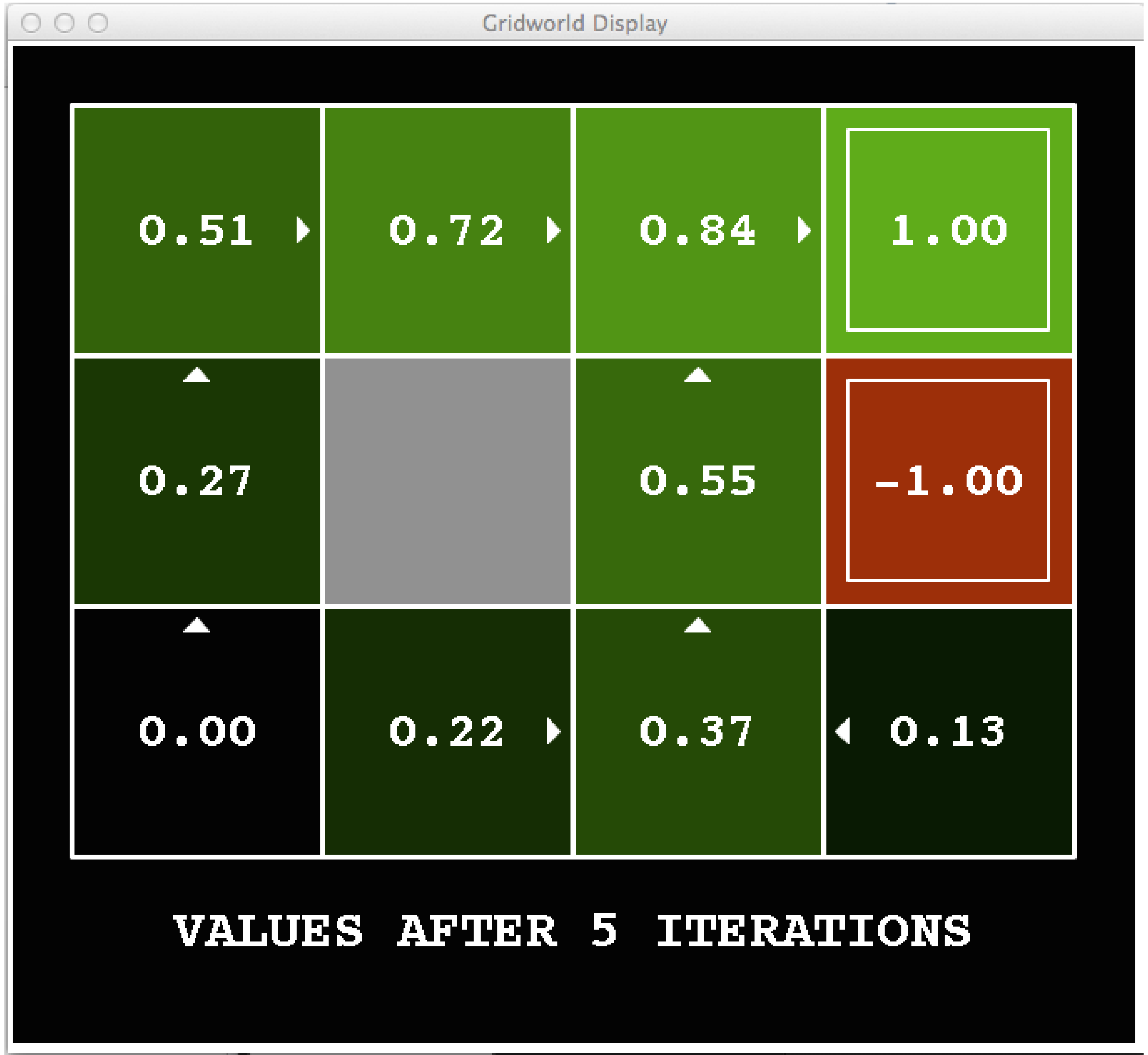
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



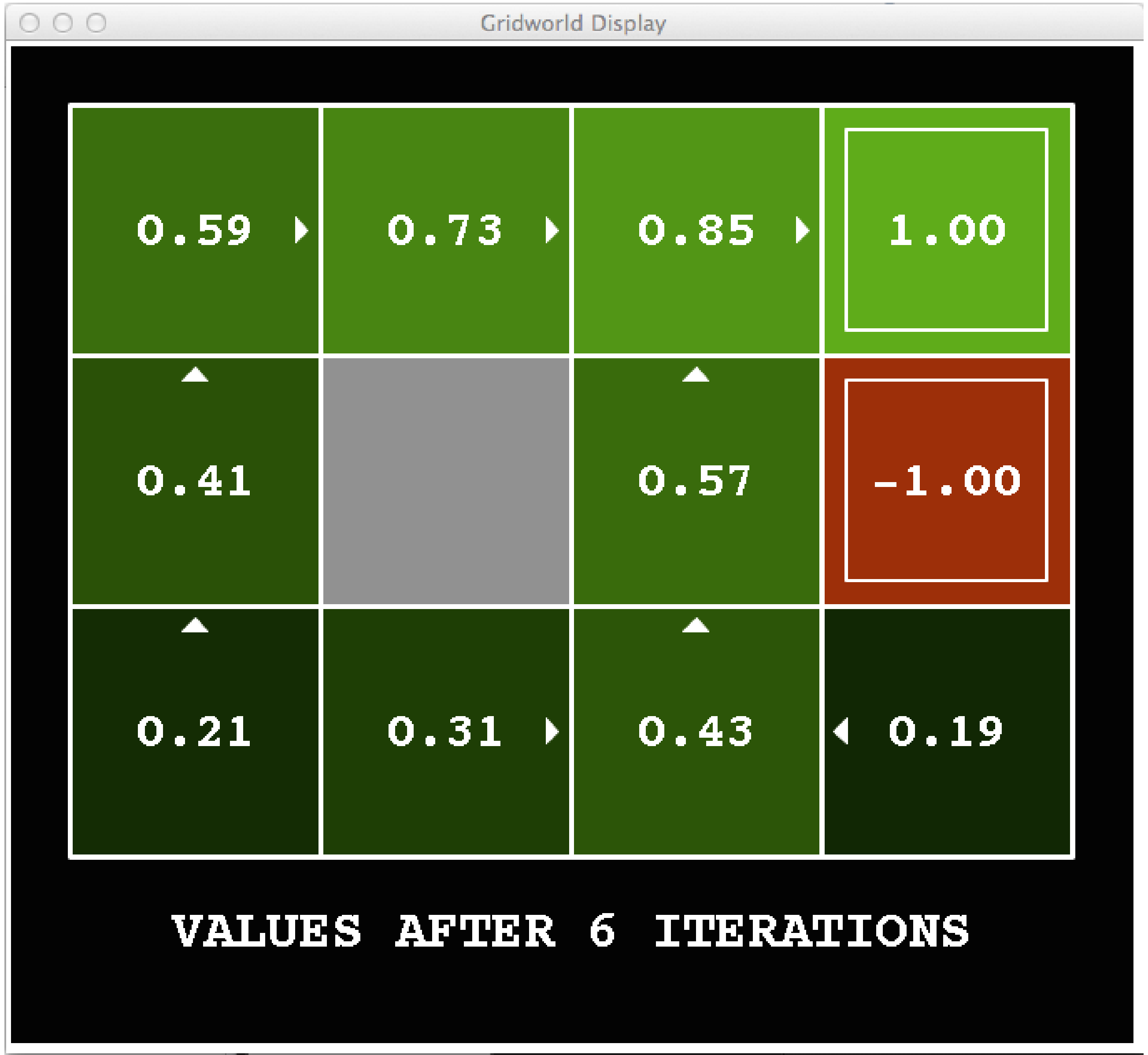
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



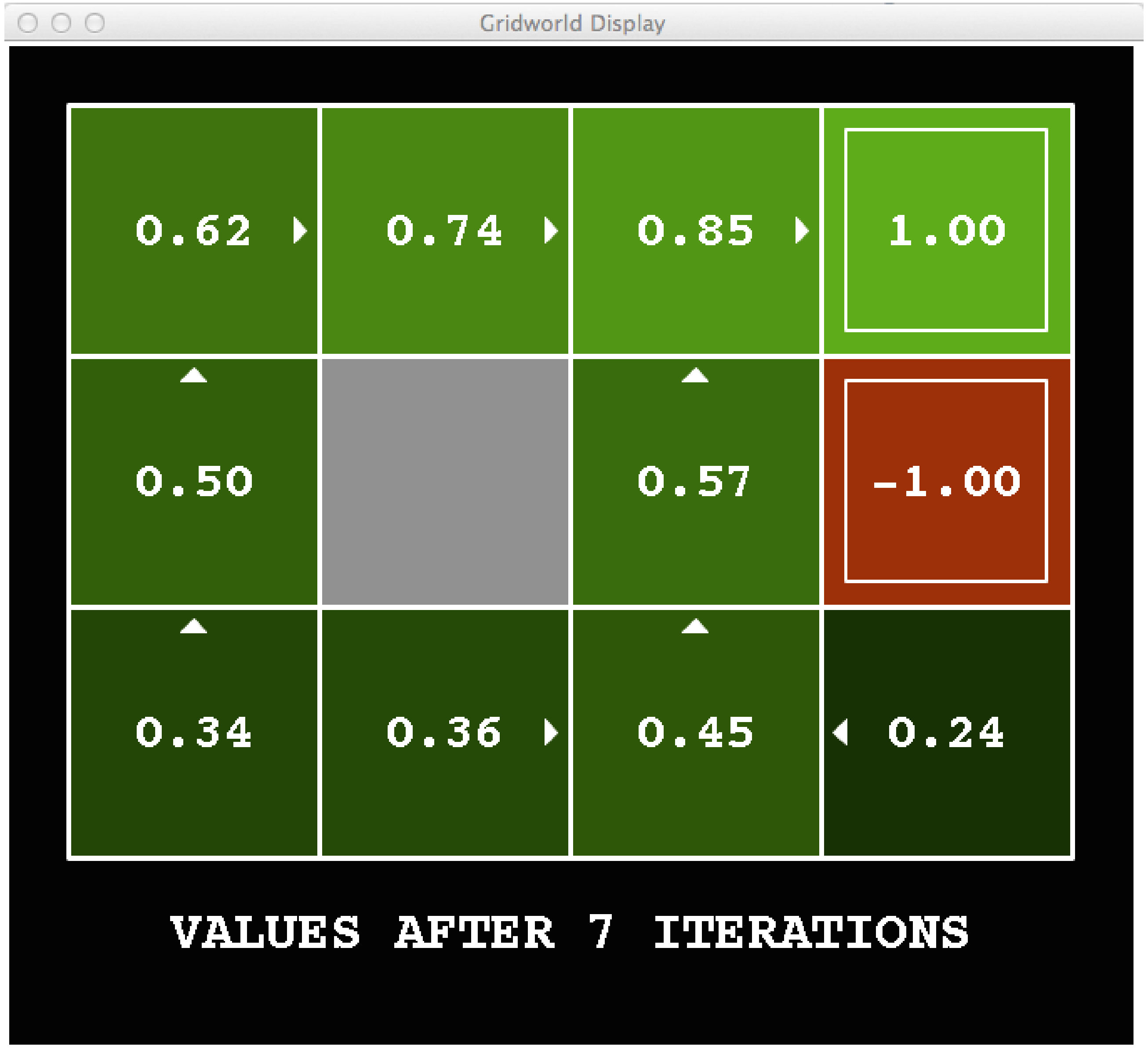
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



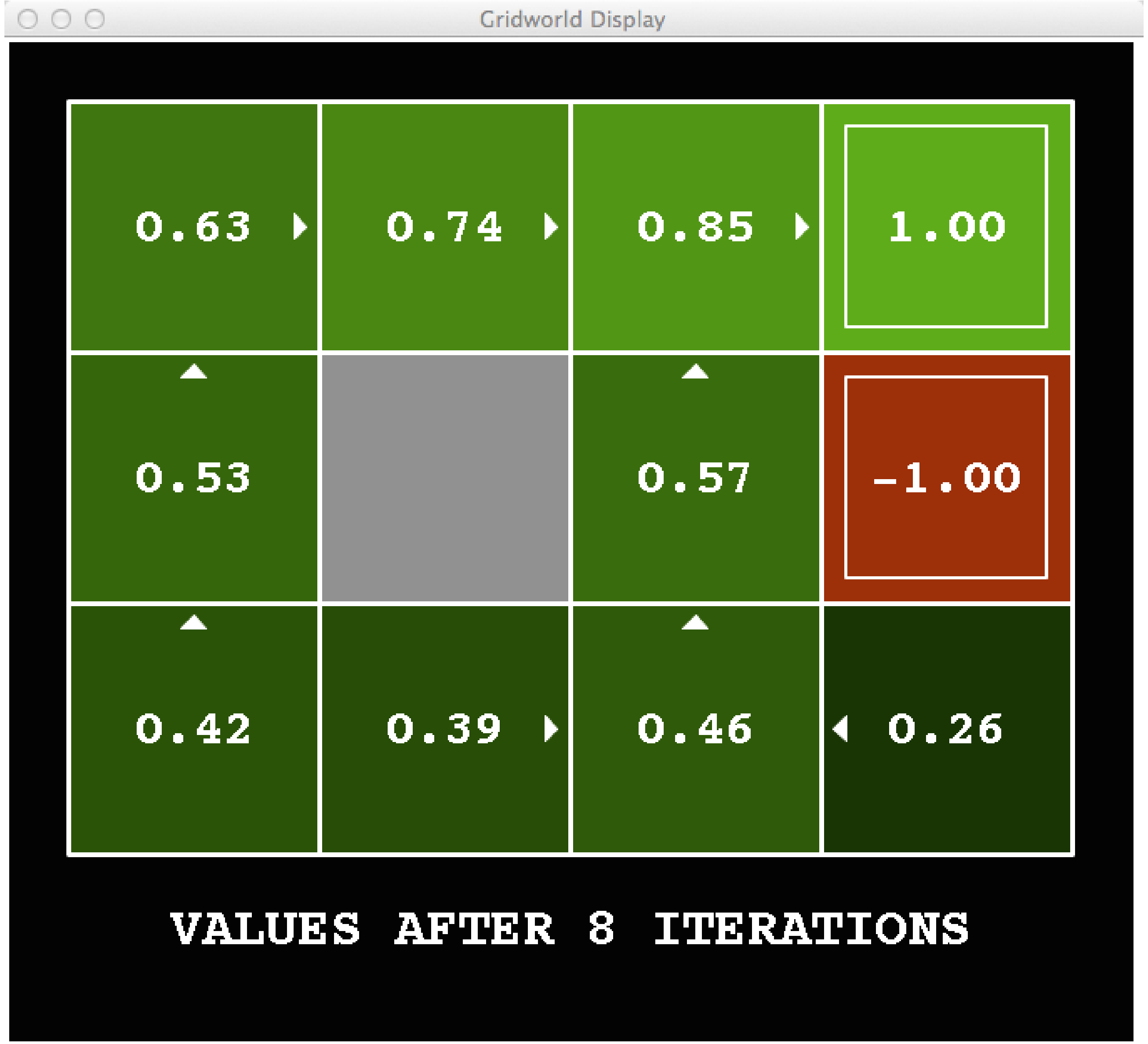
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



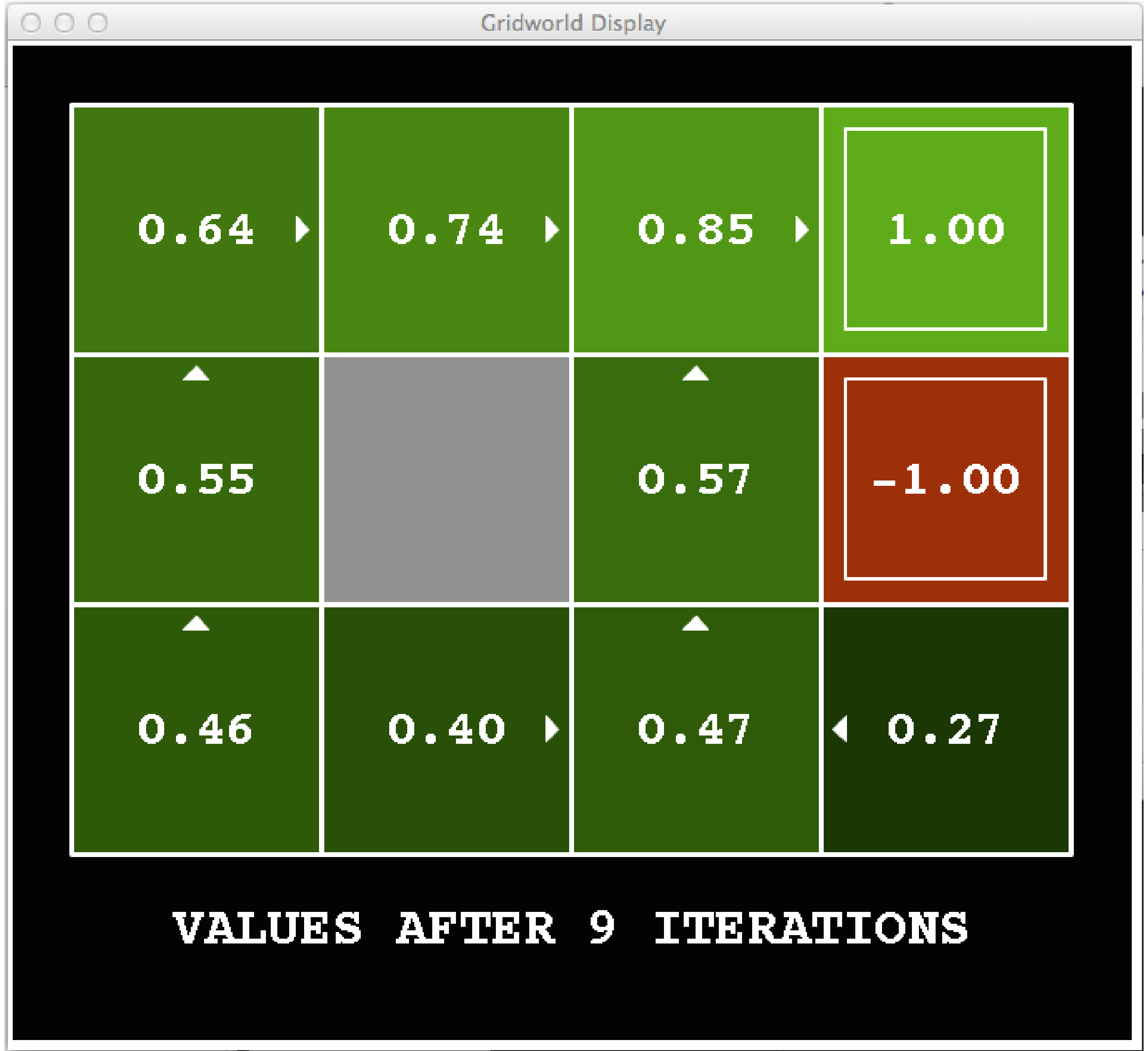
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



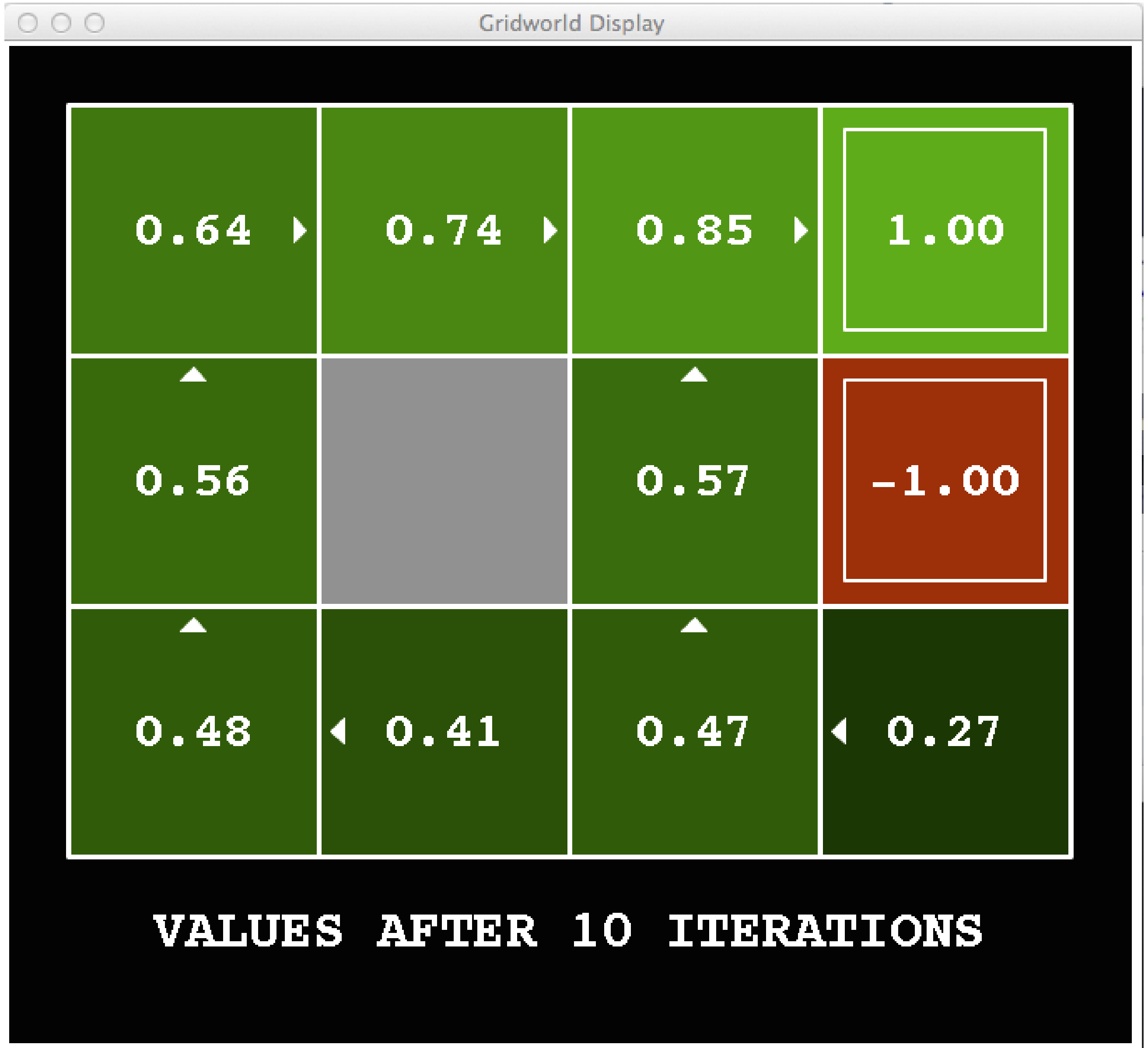
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



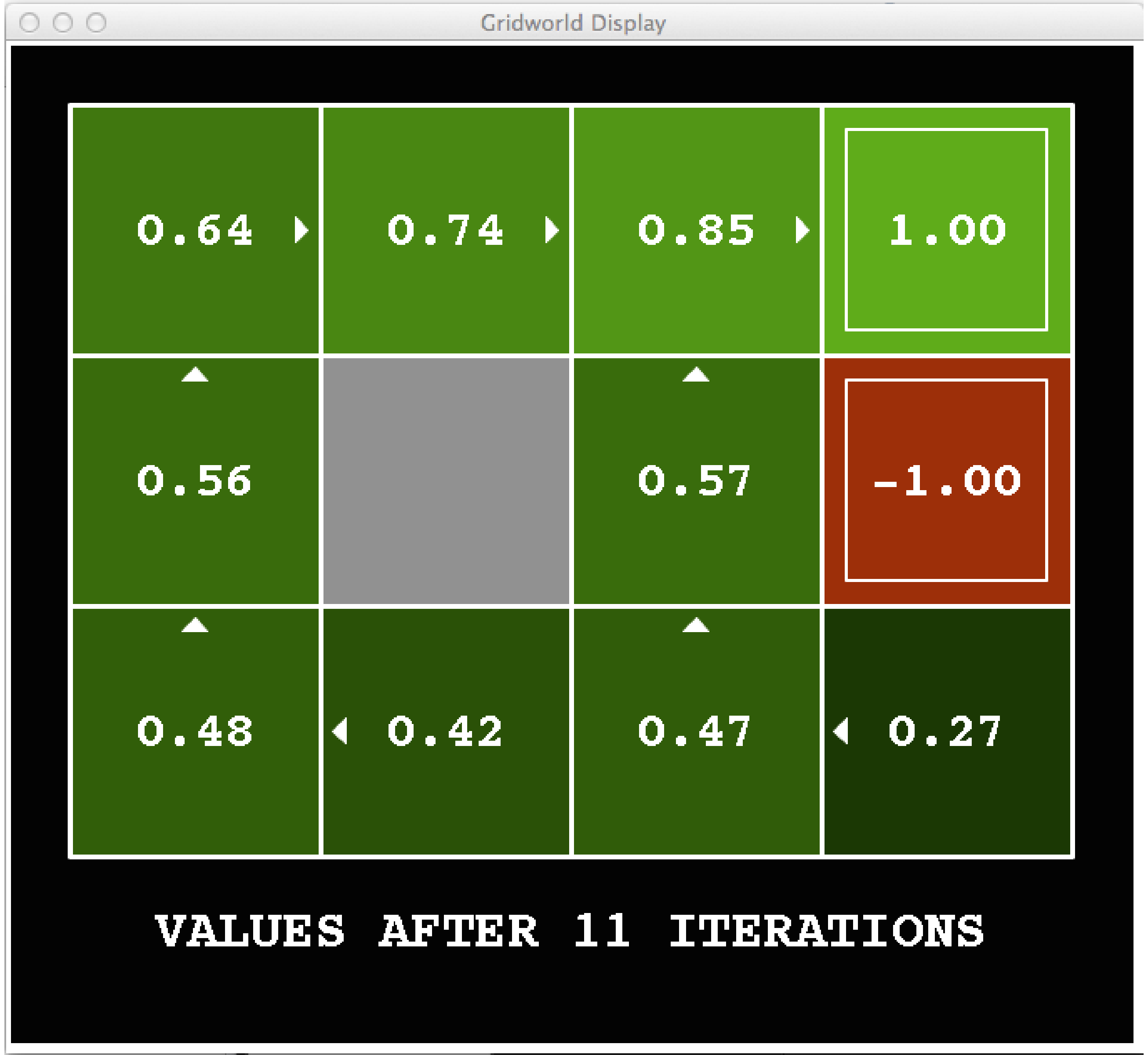
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



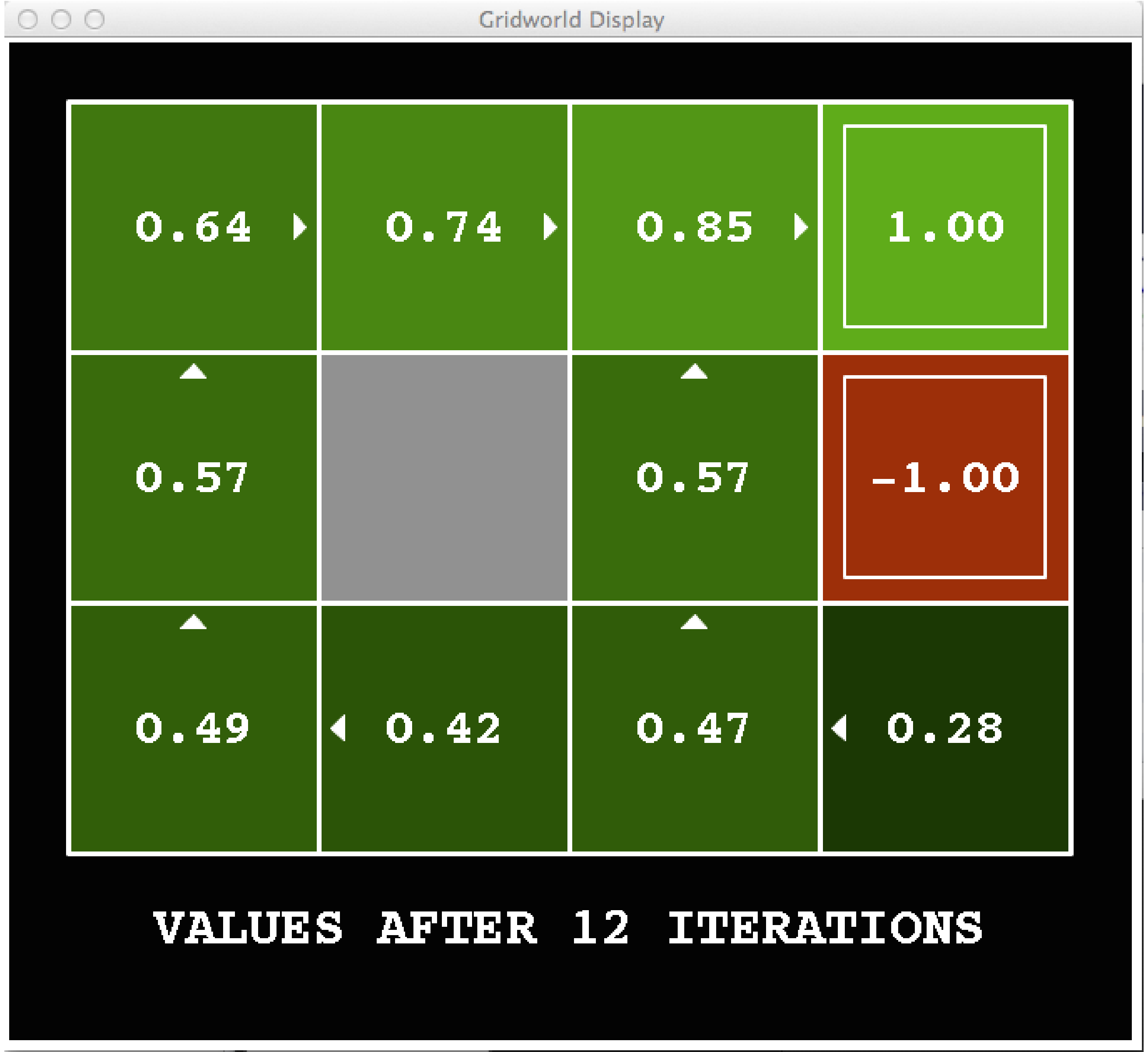
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



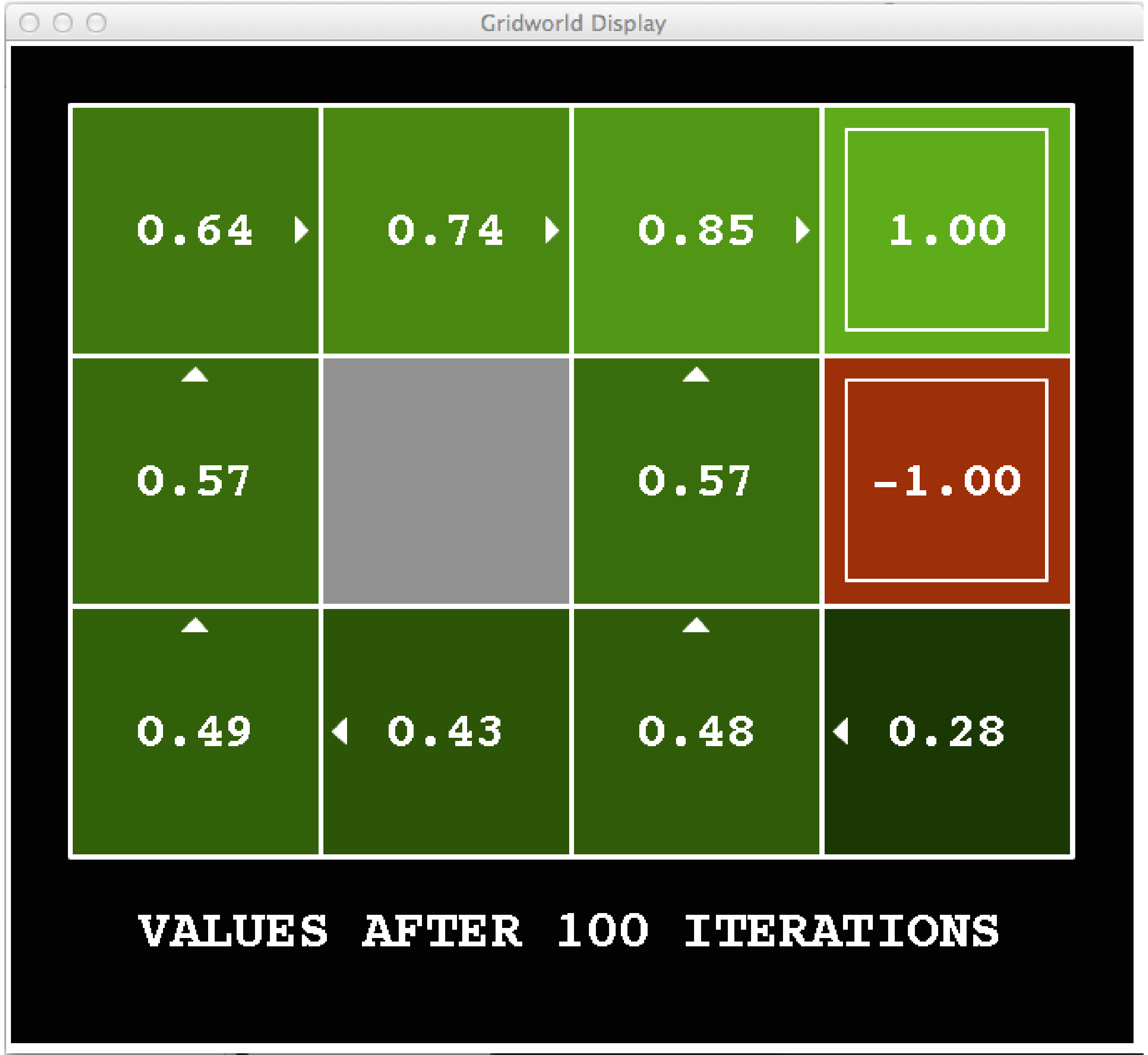
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

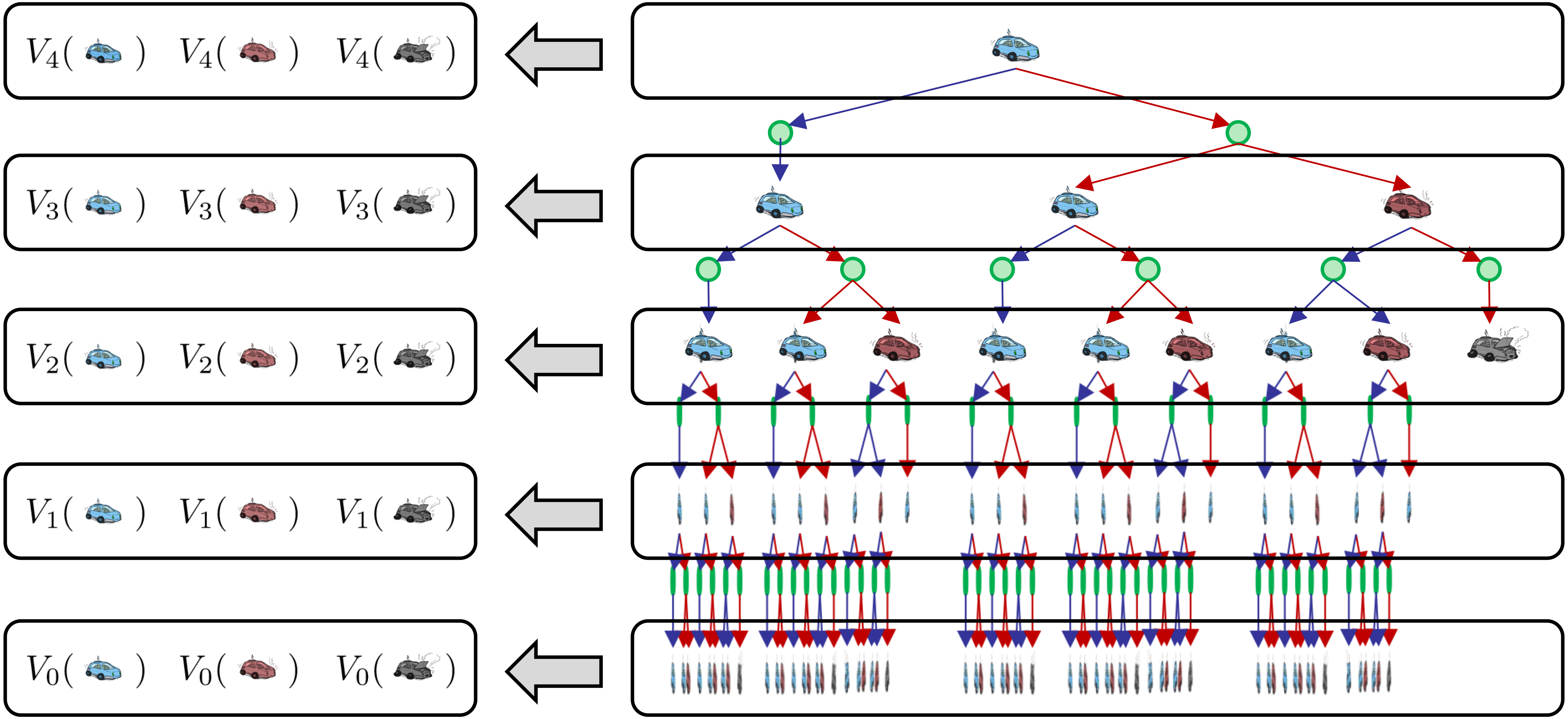
k=100



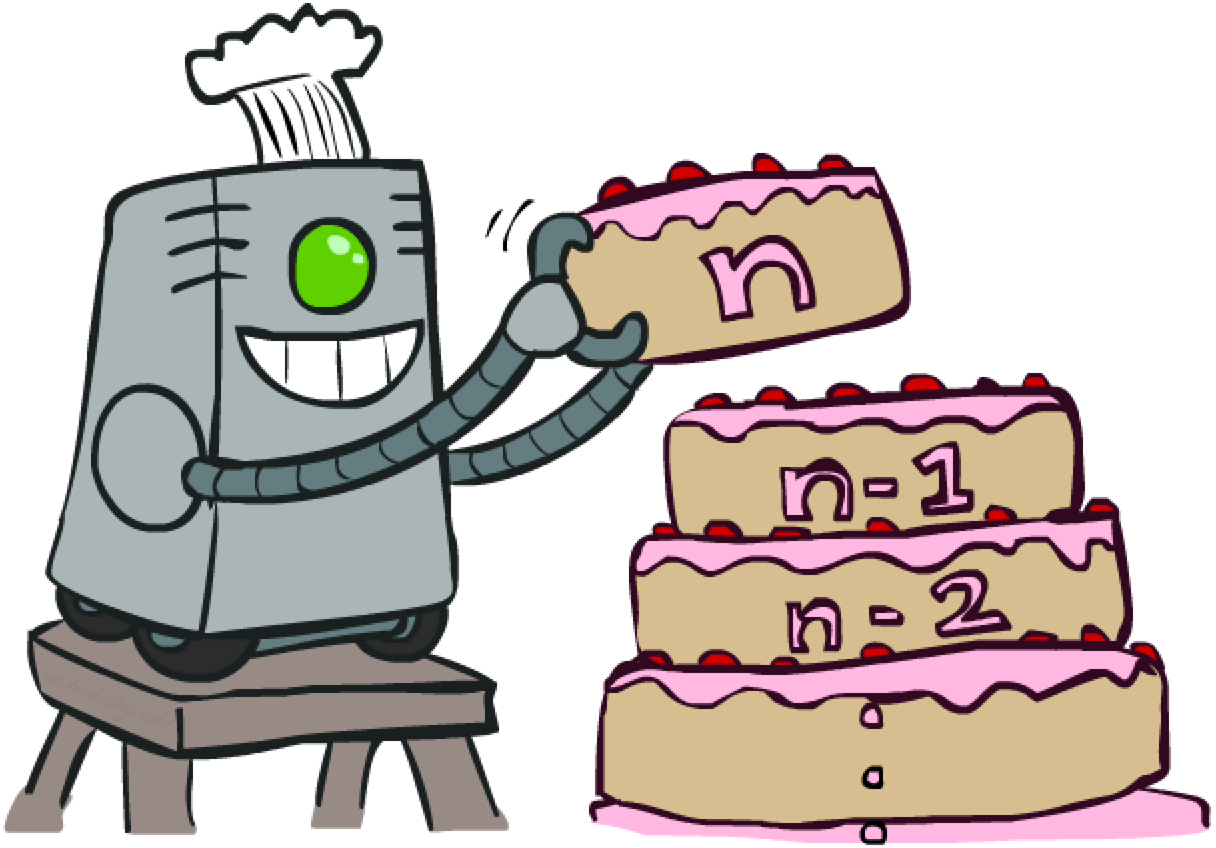
- Values converge and
- don’t change much after certain number of iterations

Noise = 0.2
Discount = 0.9
Living reward = 0

Computing Time-Limited Values (Compute v_0, v_1, v_2 , …)



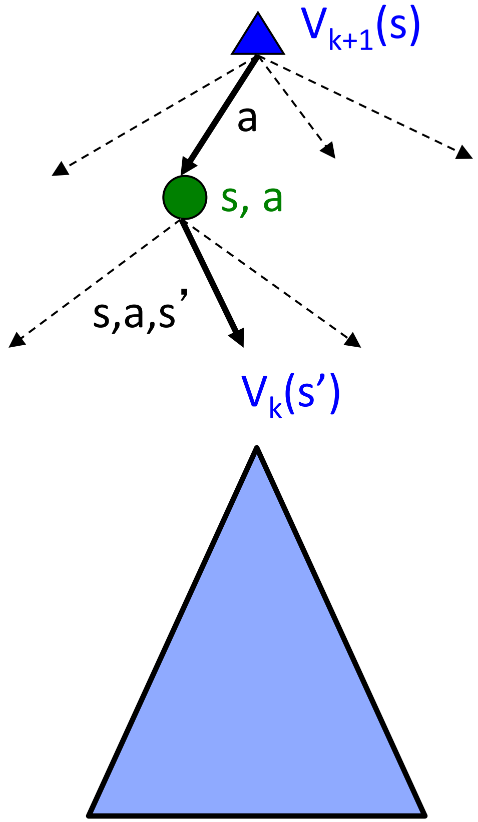
Value Iteration



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:

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

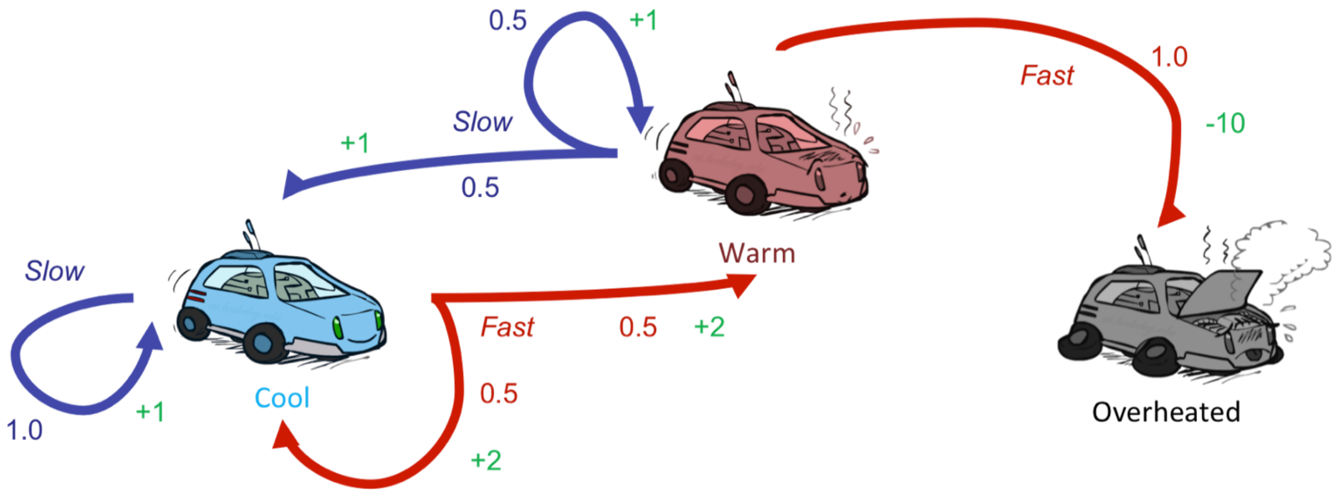


- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

Max (Q2(cool, slow) = 1+2,
Q2(cool,fast) = ((2+2)+(2+1))/2=3.5)

Example: Value Iteration

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



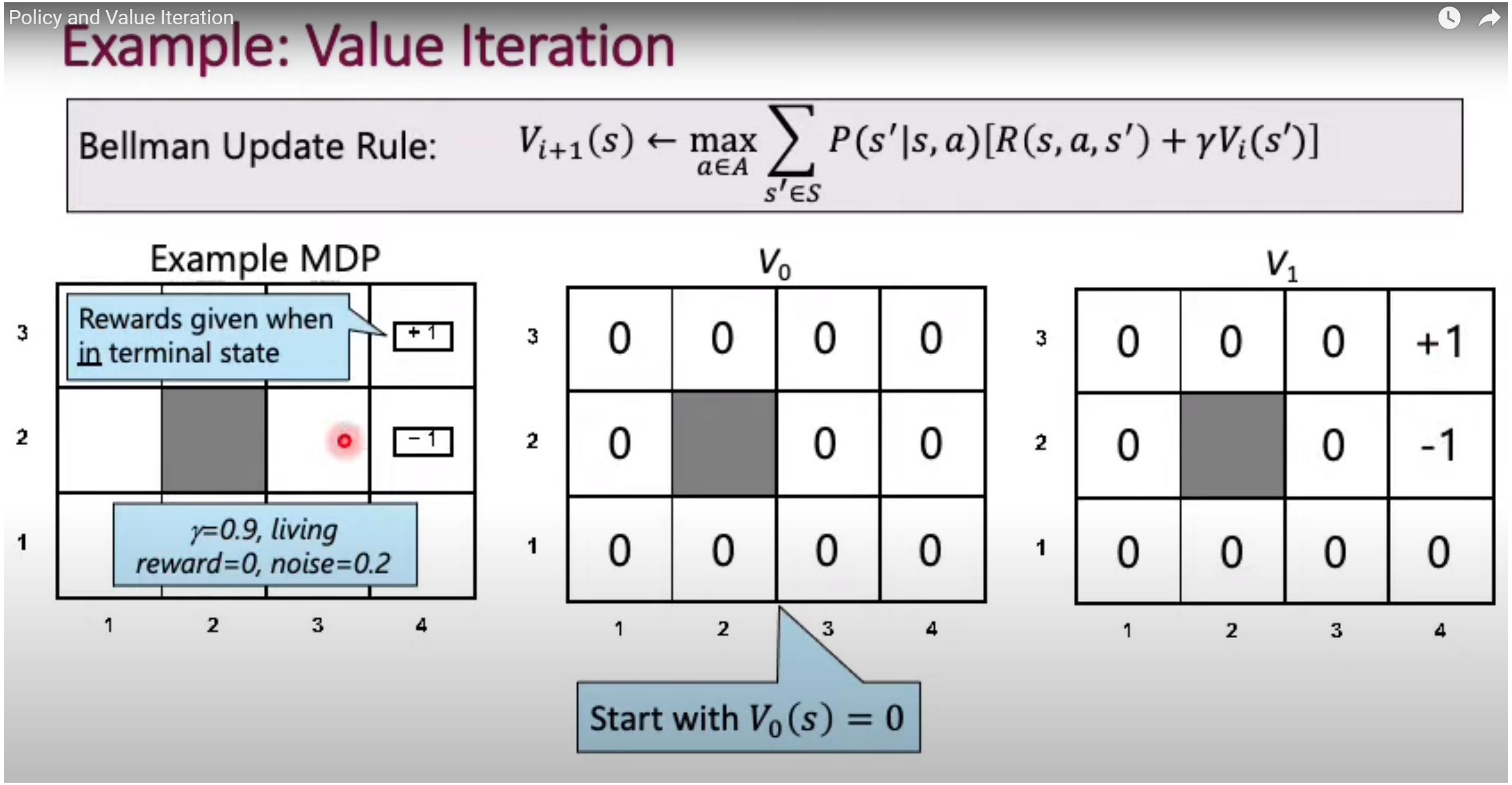
Assume no discount!

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

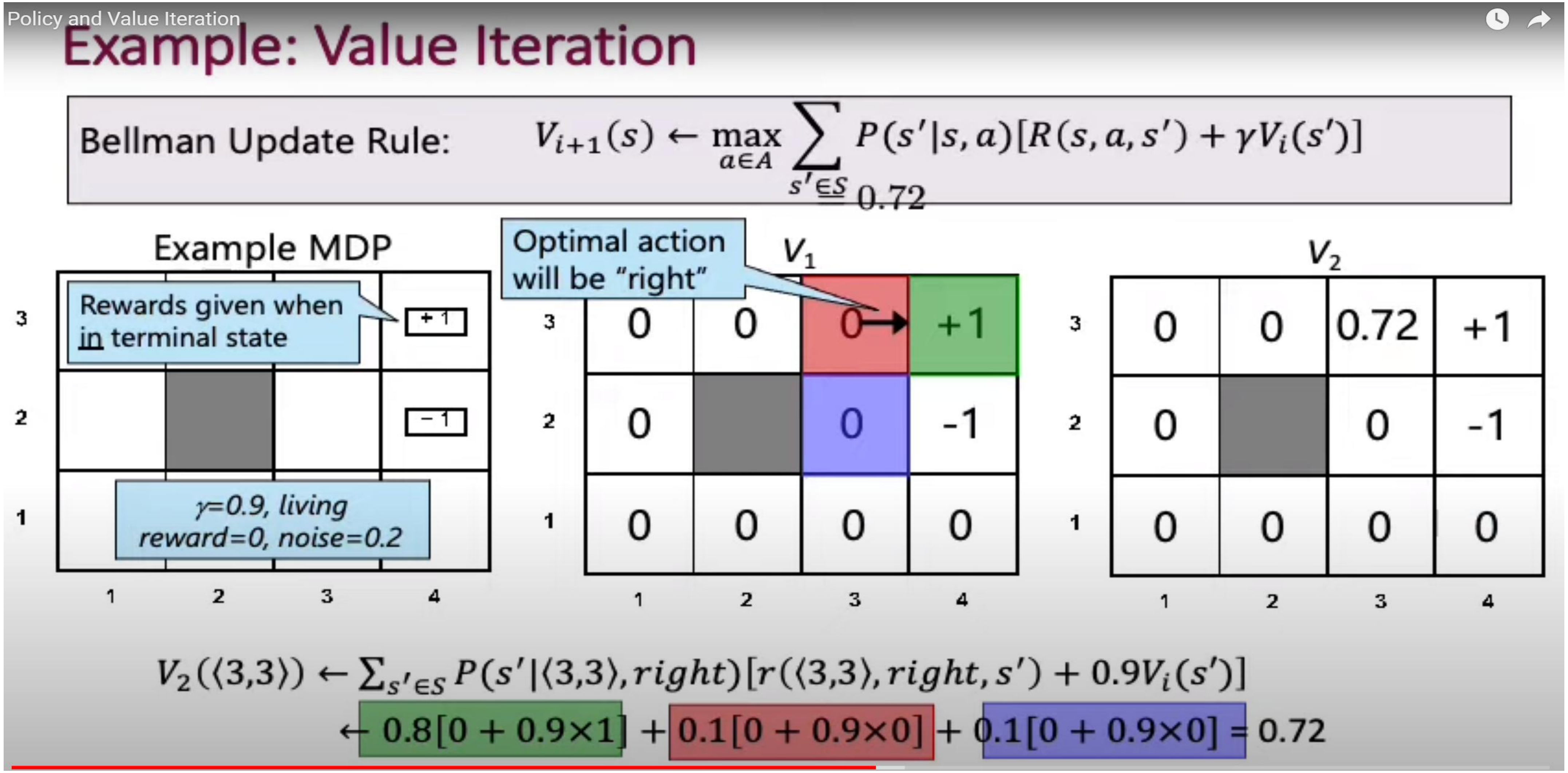
Max (Q1(cool, slow) = 1,
Q1(cool,fast) = 2))

Max (Q1(warm, slow) = 1,
Q1(warm,fast) = -10))

Example: Value Iteration



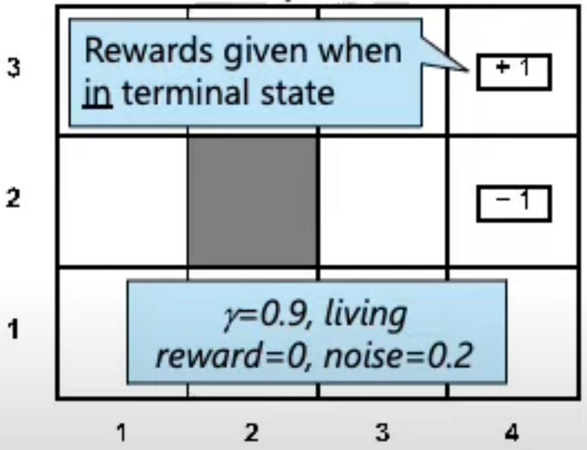
Example: Value Iteration



Example: Value Iteration

Bellman Update Rule: $$V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V_i(s')]$$

Example MDP



V_2

	1	2	3	4
3	0	0	0.72	+1
2	0		0	-1
1	0	0	0	0

V_3

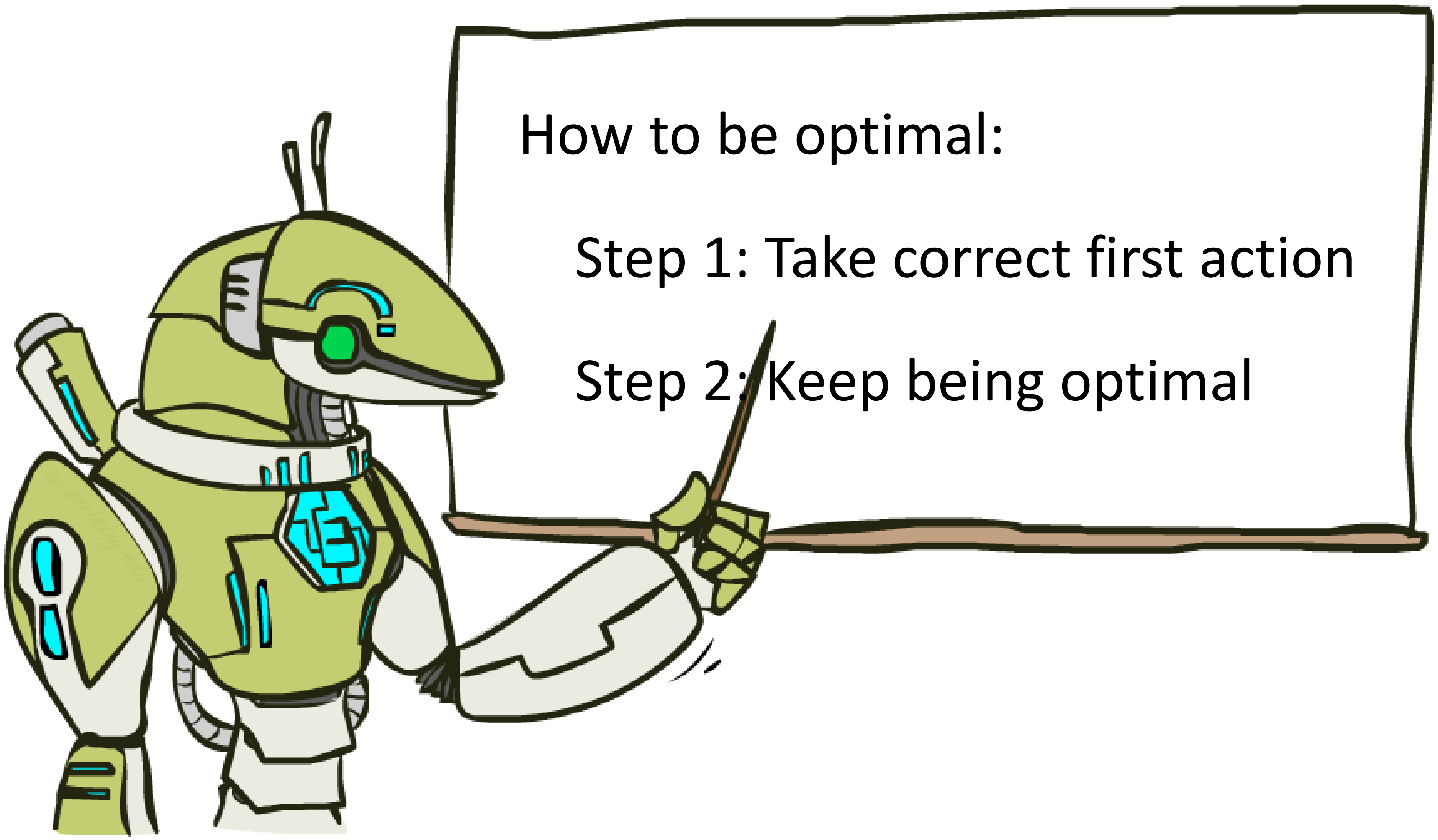
	1	2	3	4
3	0	0.52	0.78	+1
2	0		0.43	-1
1	0	0	0	0

- Information propagates outward from terminal states

GridWorld: Dynamic Programming Demo

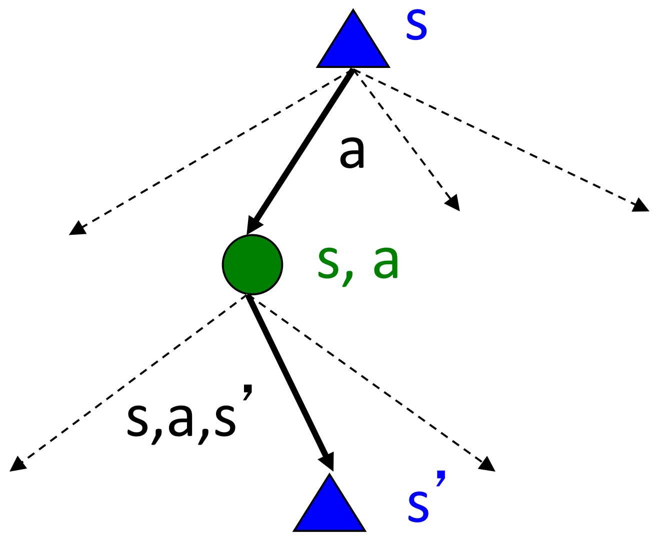
- https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html

The Bellman Equations



The Bellman Equations

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



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

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

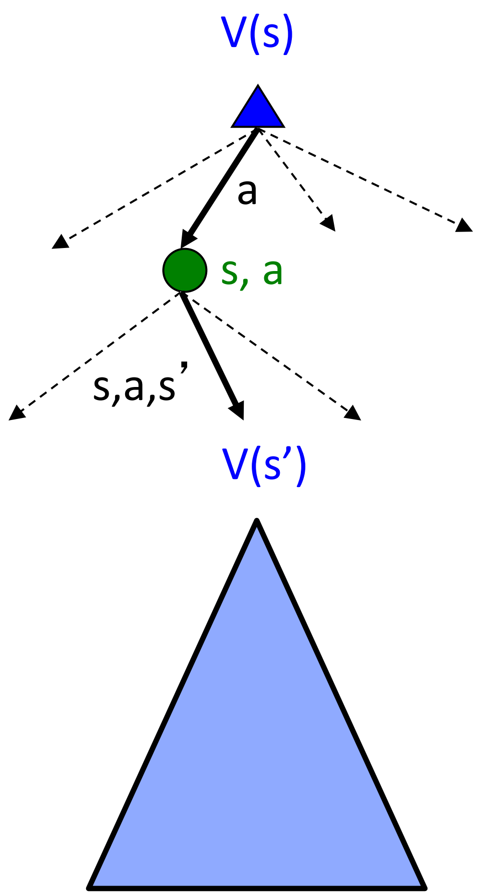
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

- These are the Bellman equations, and they characterize optimal values in a way we'll use over and over.
- The Bellman equation is the basis of the value iteration algorithm for solving MDPs.

Value Iteration

- Bellman equations characterize the optimal values:

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



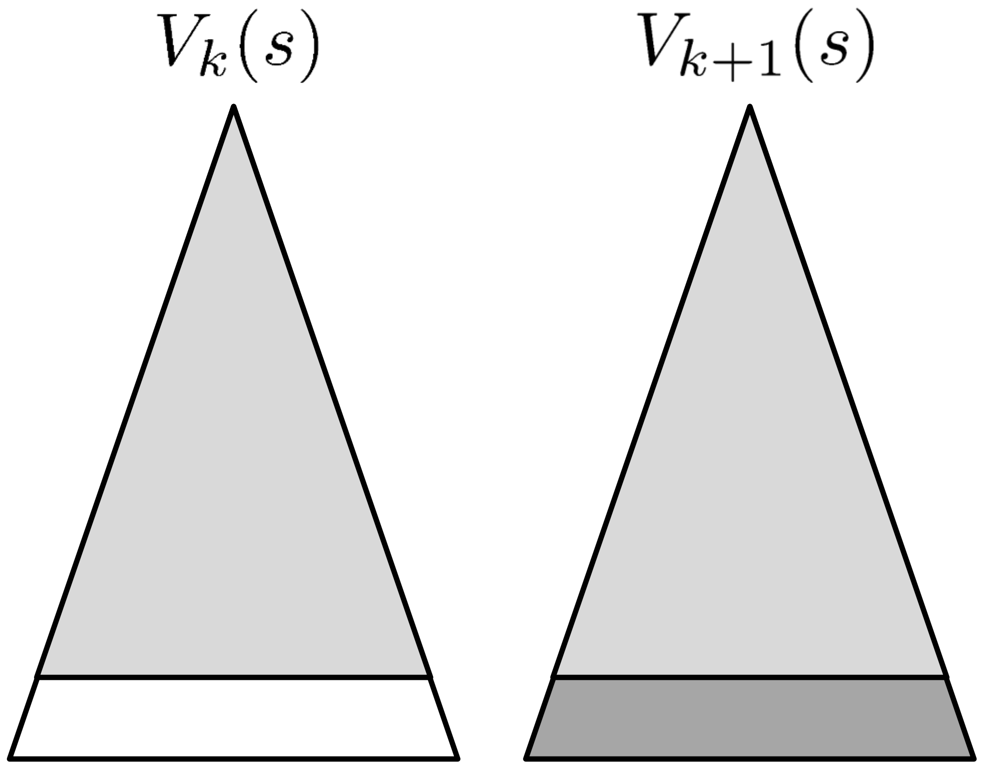
- Value iteration computes them:

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

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

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



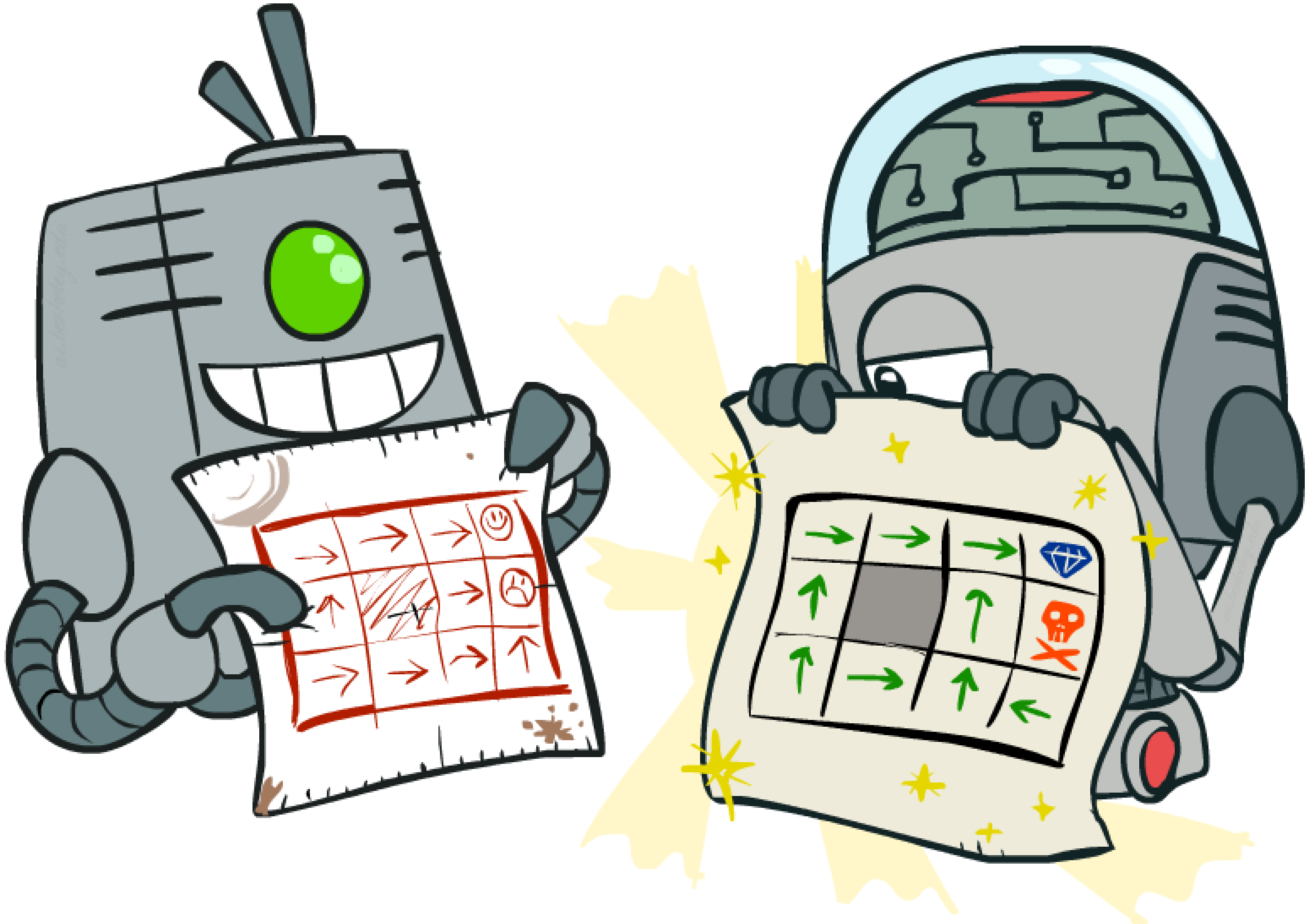
Value iteration algorithm

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s, a, s')$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum relative change in the utility of any state

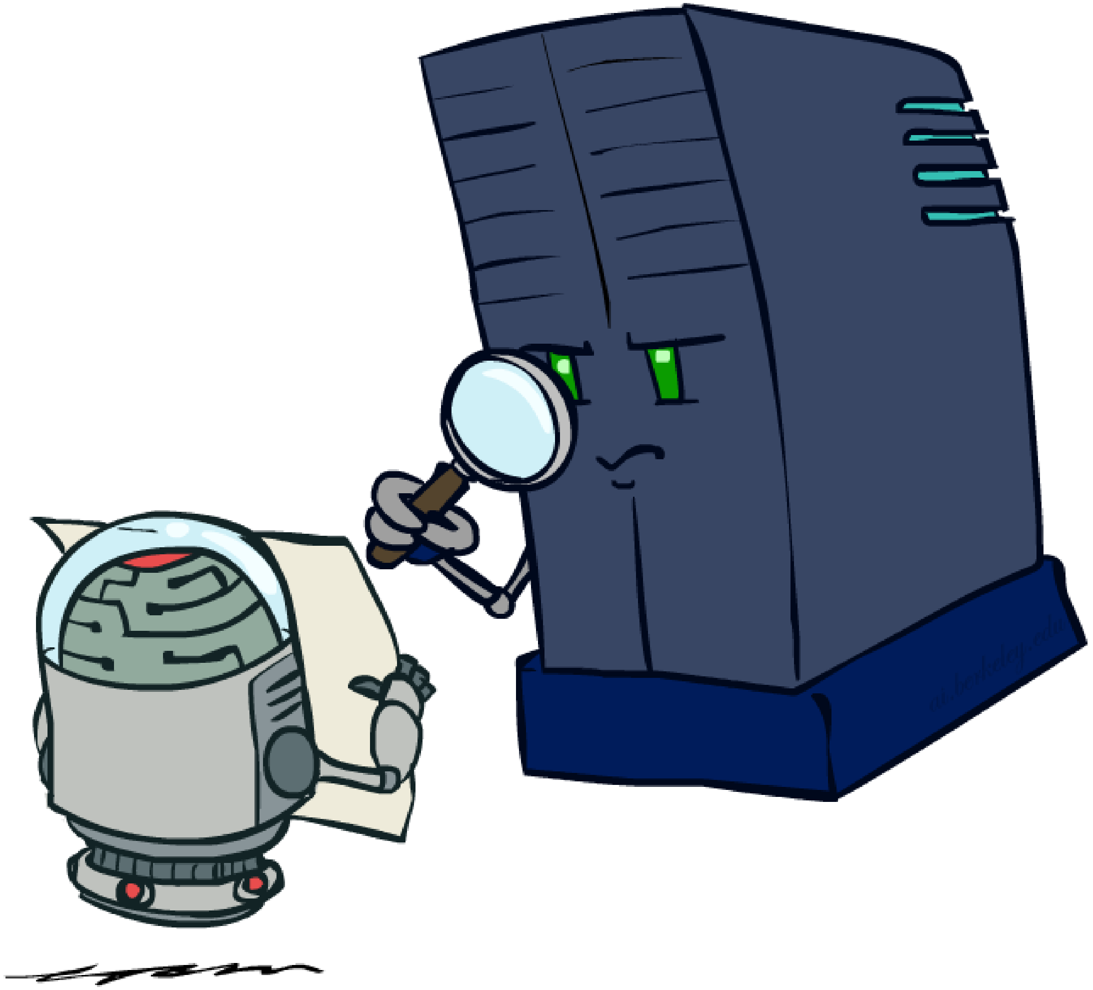
  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s in S **do**
      $U'[s] \leftarrow \max_{a \in A(s)}$ Q-VALUE(mdp, s, a, U)
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return U

Figure 16.6 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (16.12).

Policy Methods

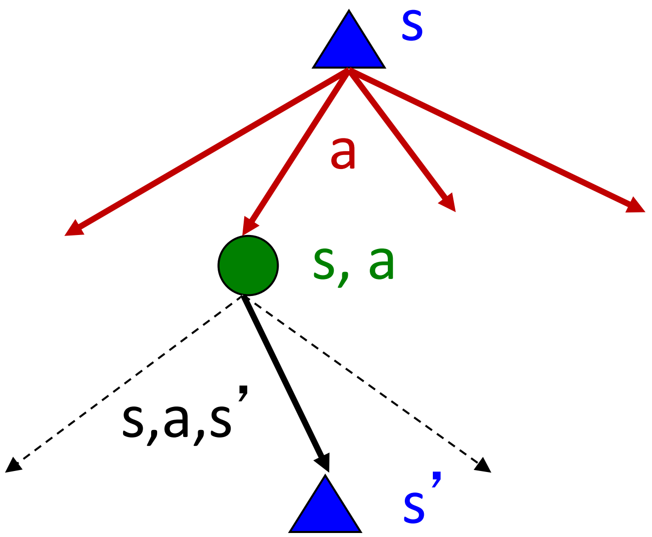


Policy Evaluation

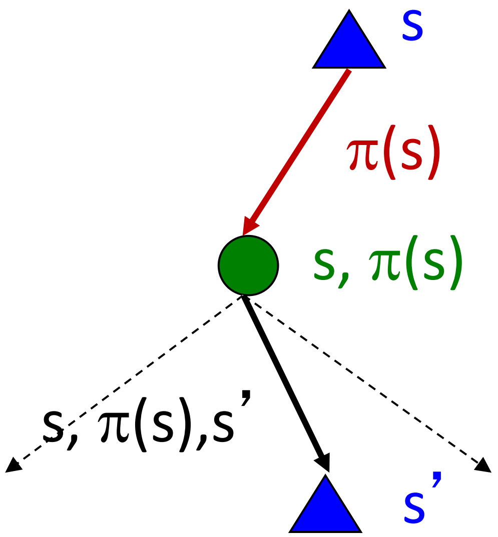


Fixed Policies

Do the optimal action



Do what π says to do



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

Utilities for a Fixed Policy

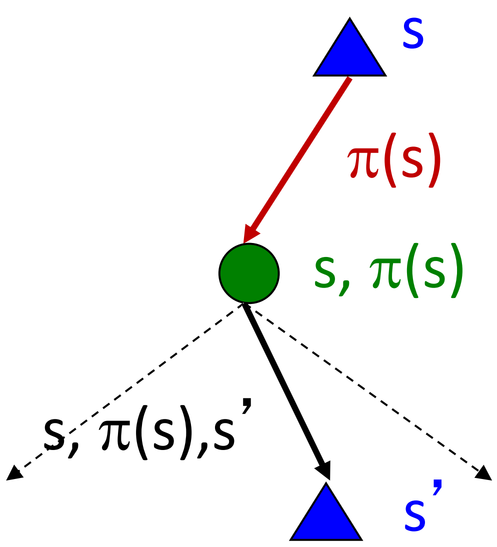
- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

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

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

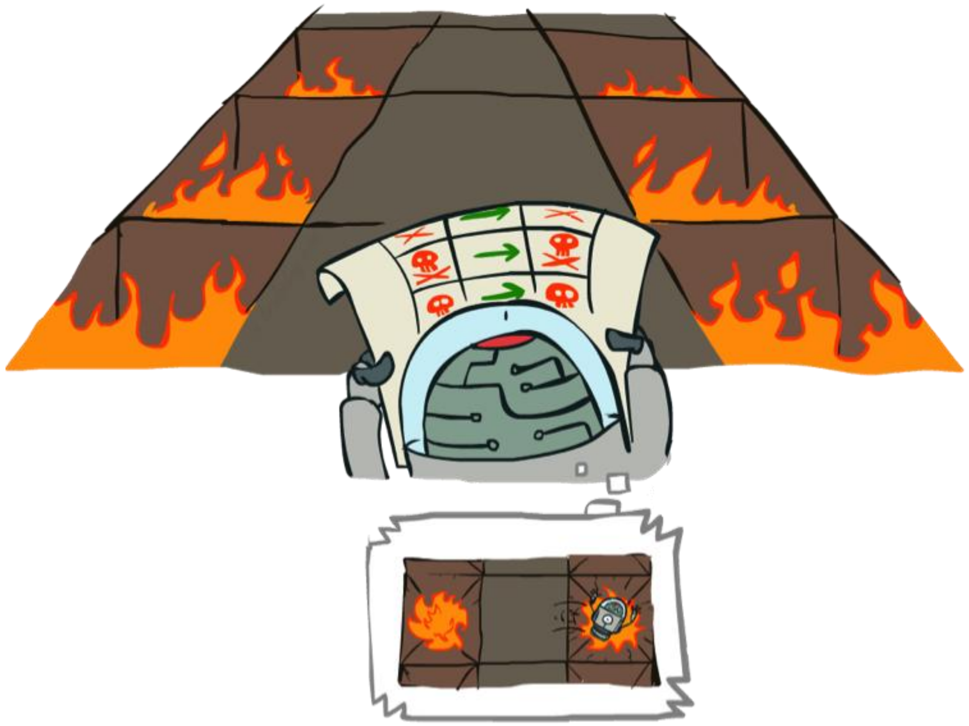
- Recursive relation (one-step look-ahead / Bellman equation):

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

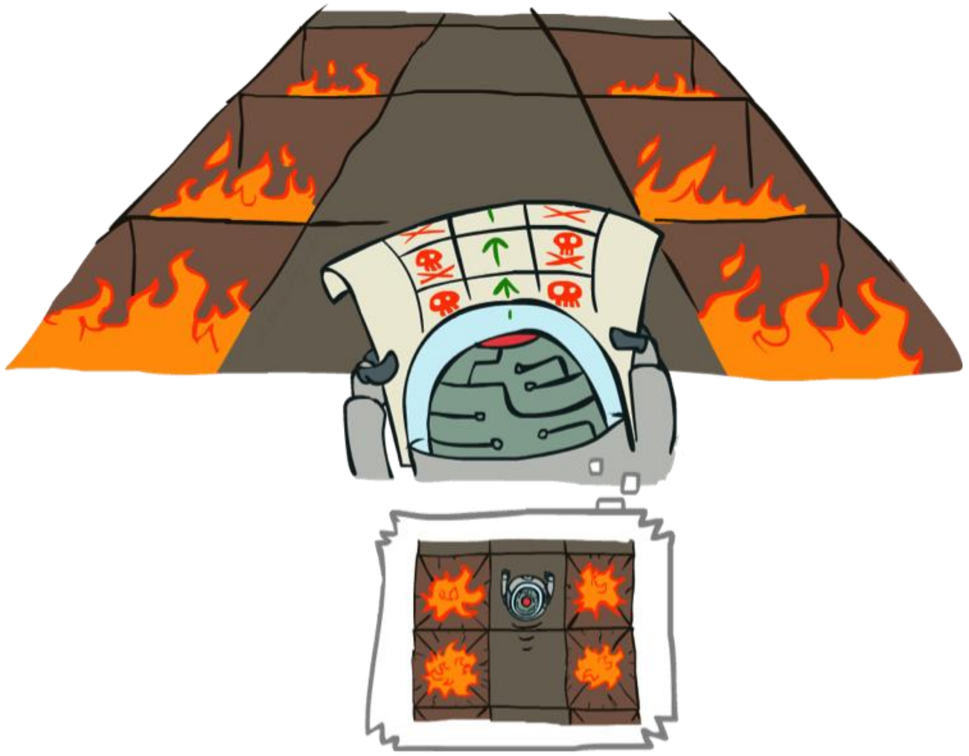


Example: Policy Evaluation

Always Go Right



Always Go Forward



Example: Policy Evaluation

Always Go Right

Bad Policy

Always Go Forward

Good Policy

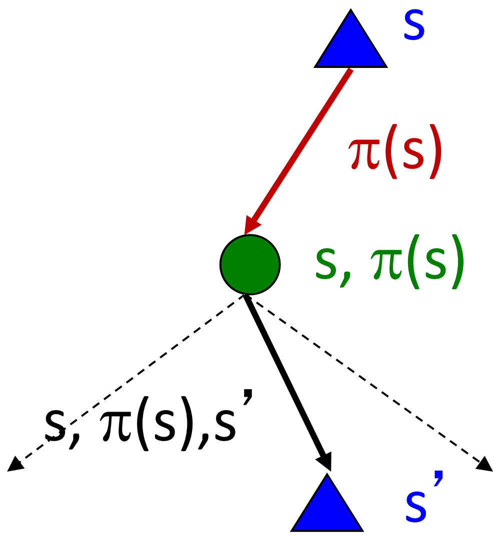
Policy Evaluation

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

- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

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

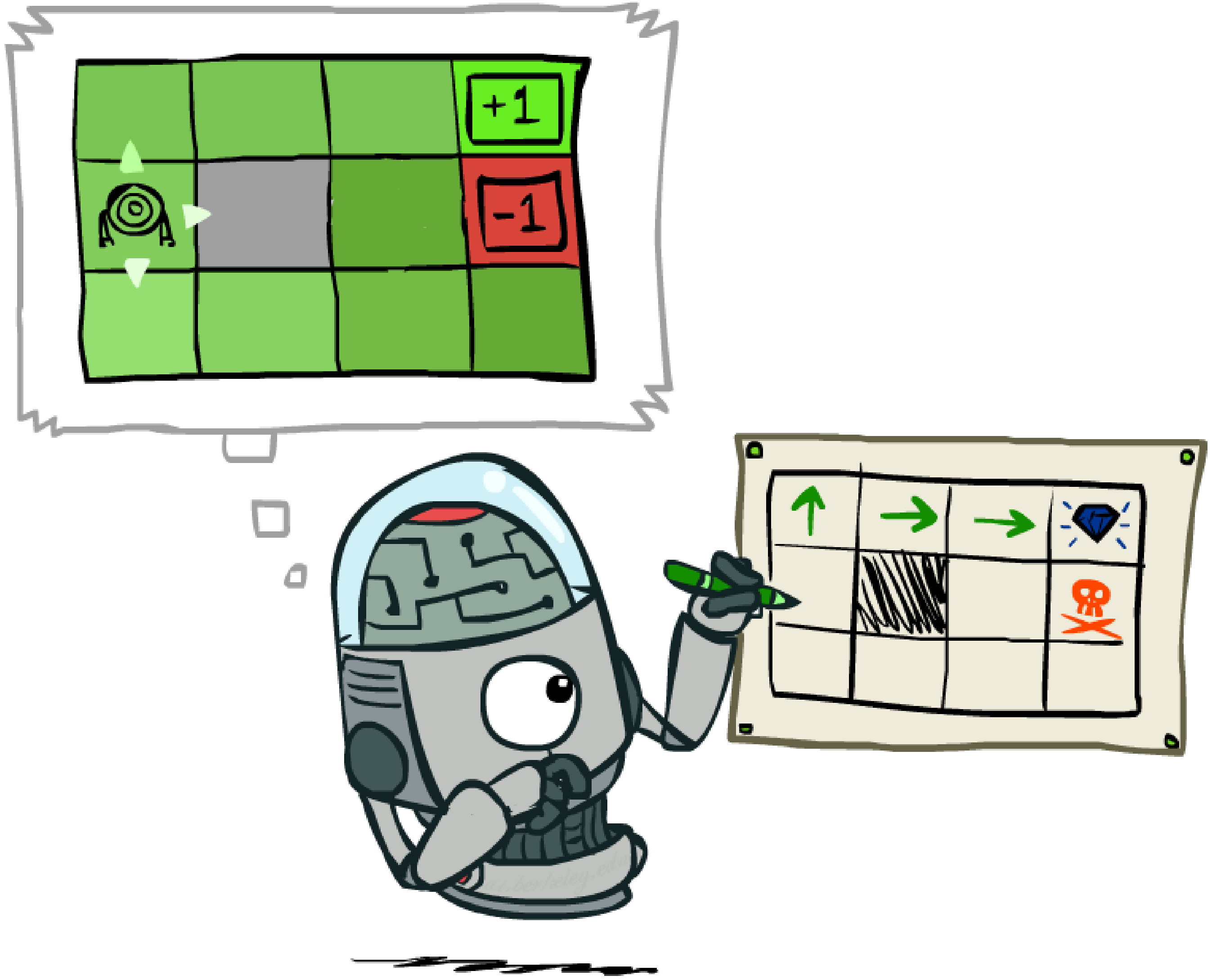
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$



- Efficiency: $O(S^2)$ per iteration

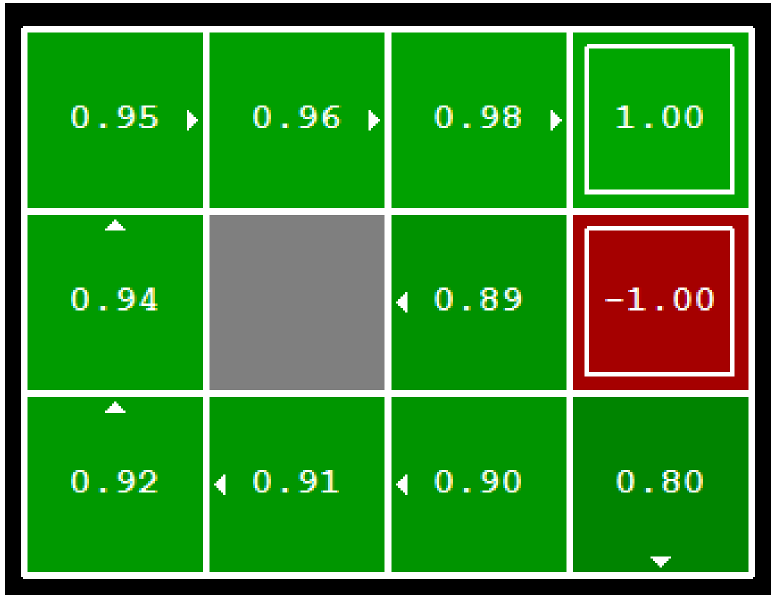
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)

Policy Extraction



Computing Actions from Values

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

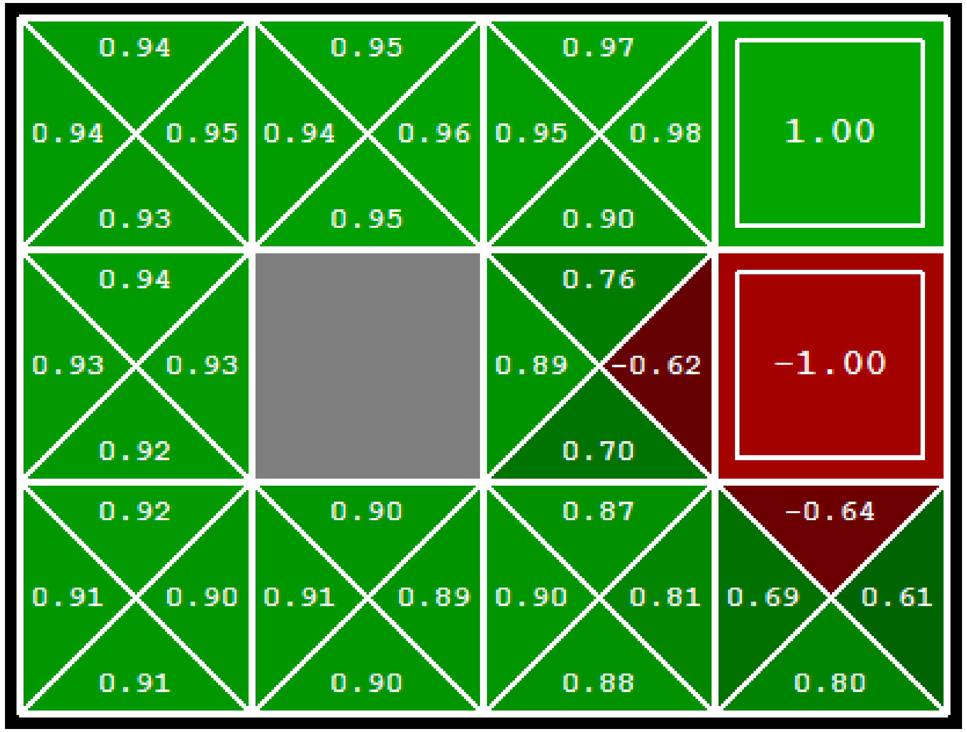


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

Computing Actions from Q-Values

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

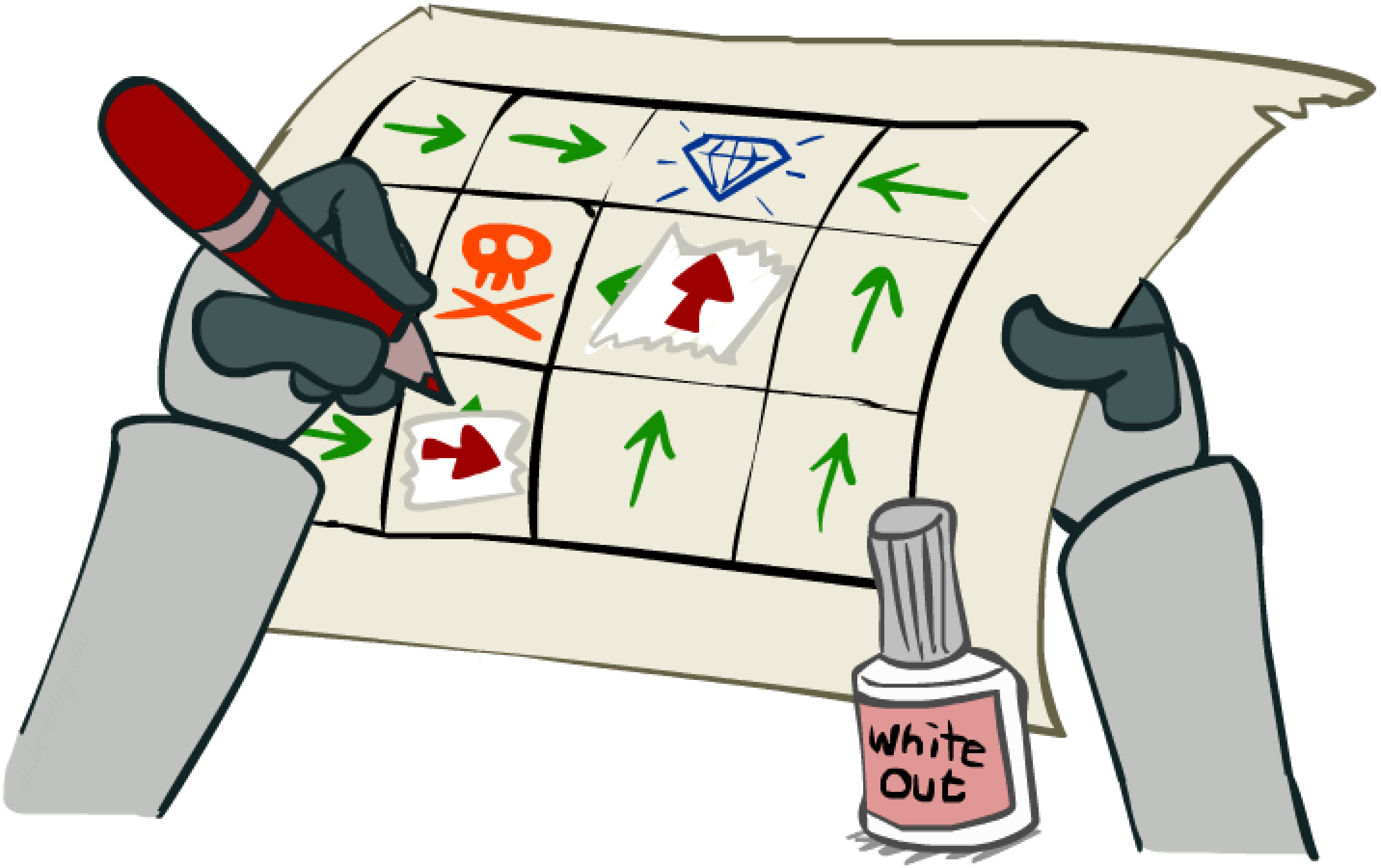


- How should we act?
  - Completely trivial to decide!

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

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

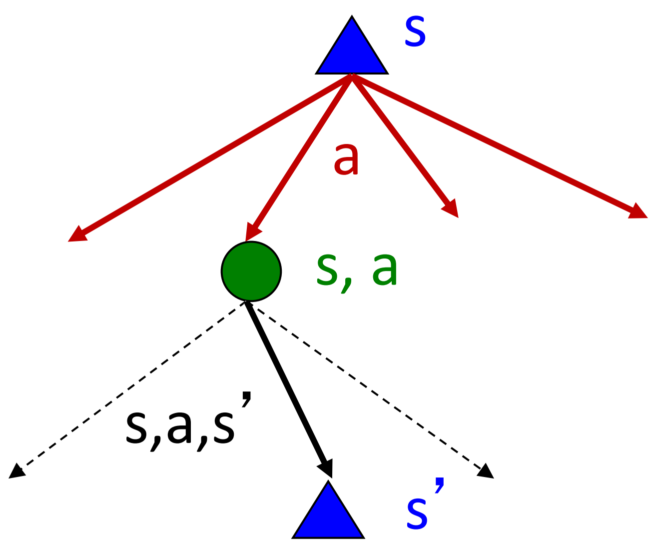
Policy Iteration



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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



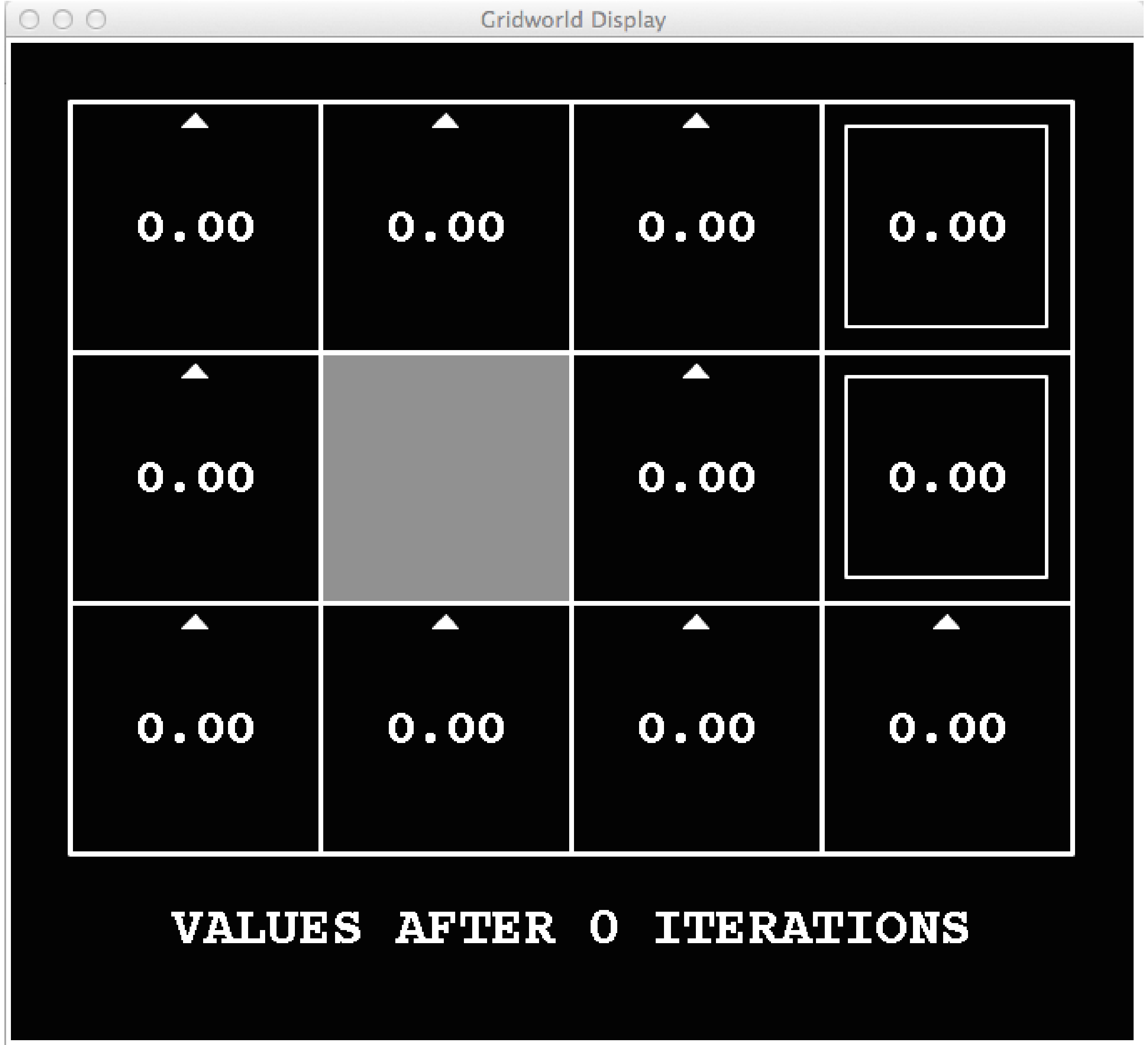
- Problem 1: It’s slow – O(S^2A) per iteration

- Problem 2: The “max” at each state rarely changes

- Problem 3: The policy often converges long before the values

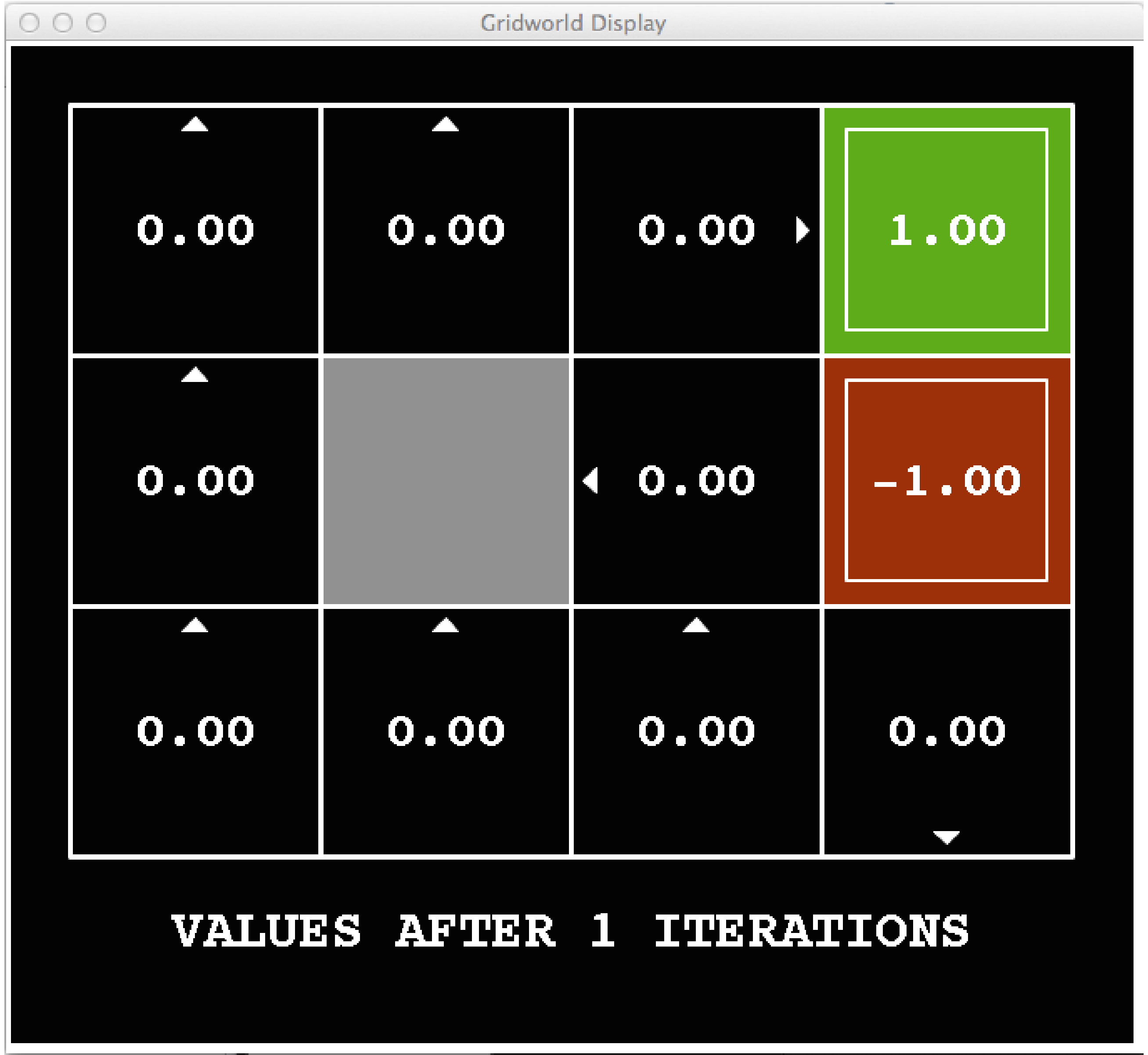
[Demo: value iteration (L9D2)]

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

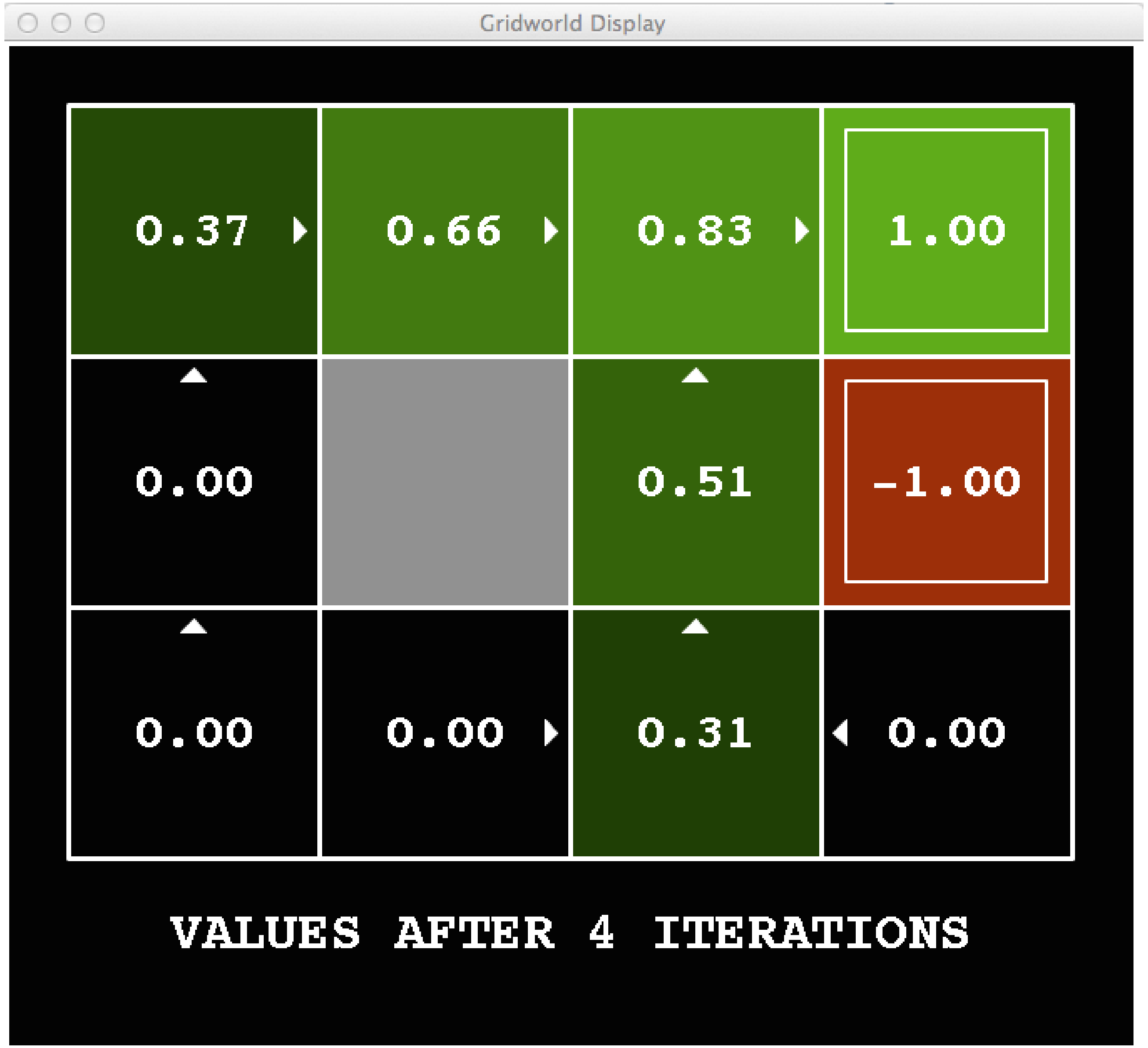
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

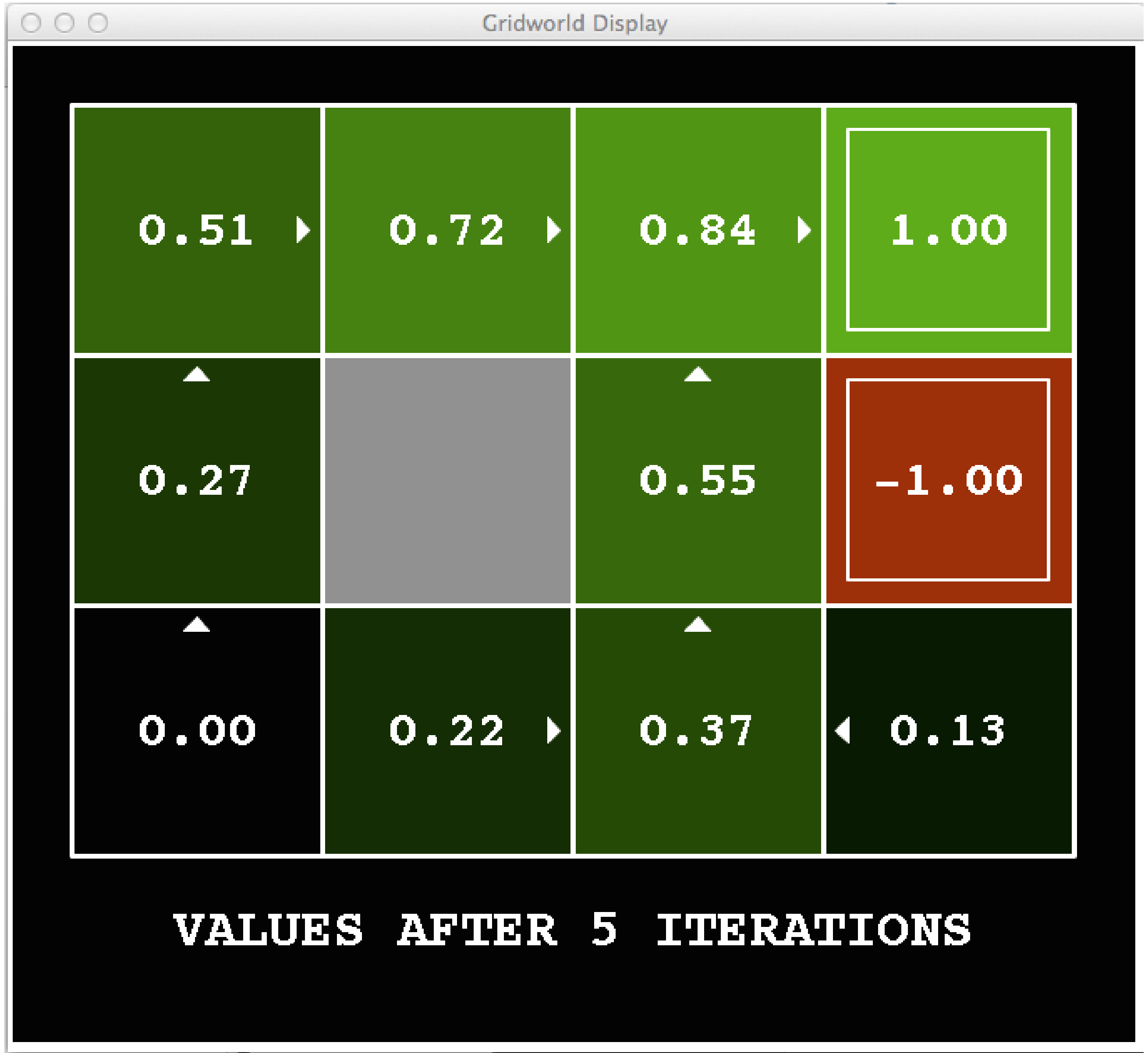
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



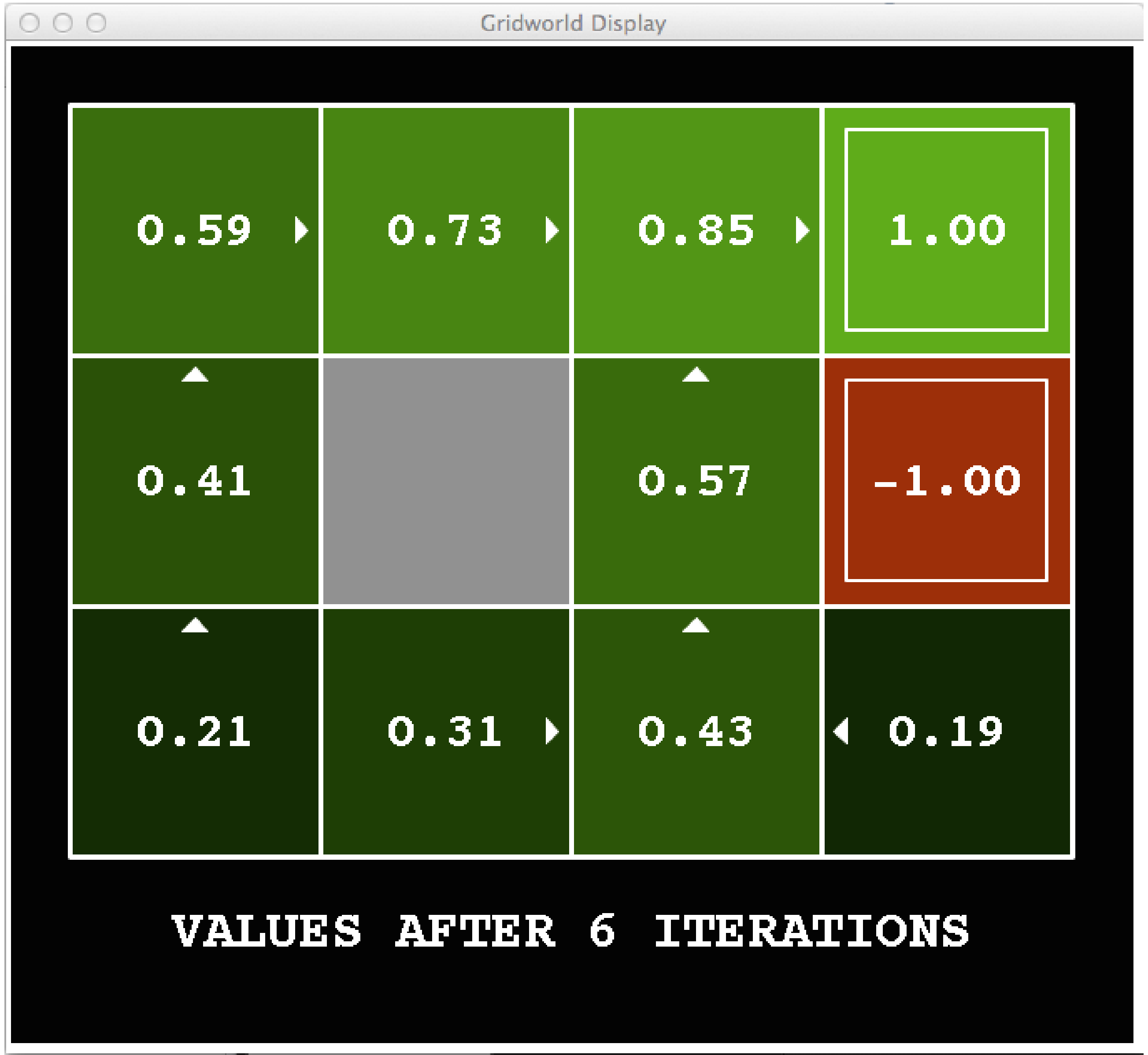
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



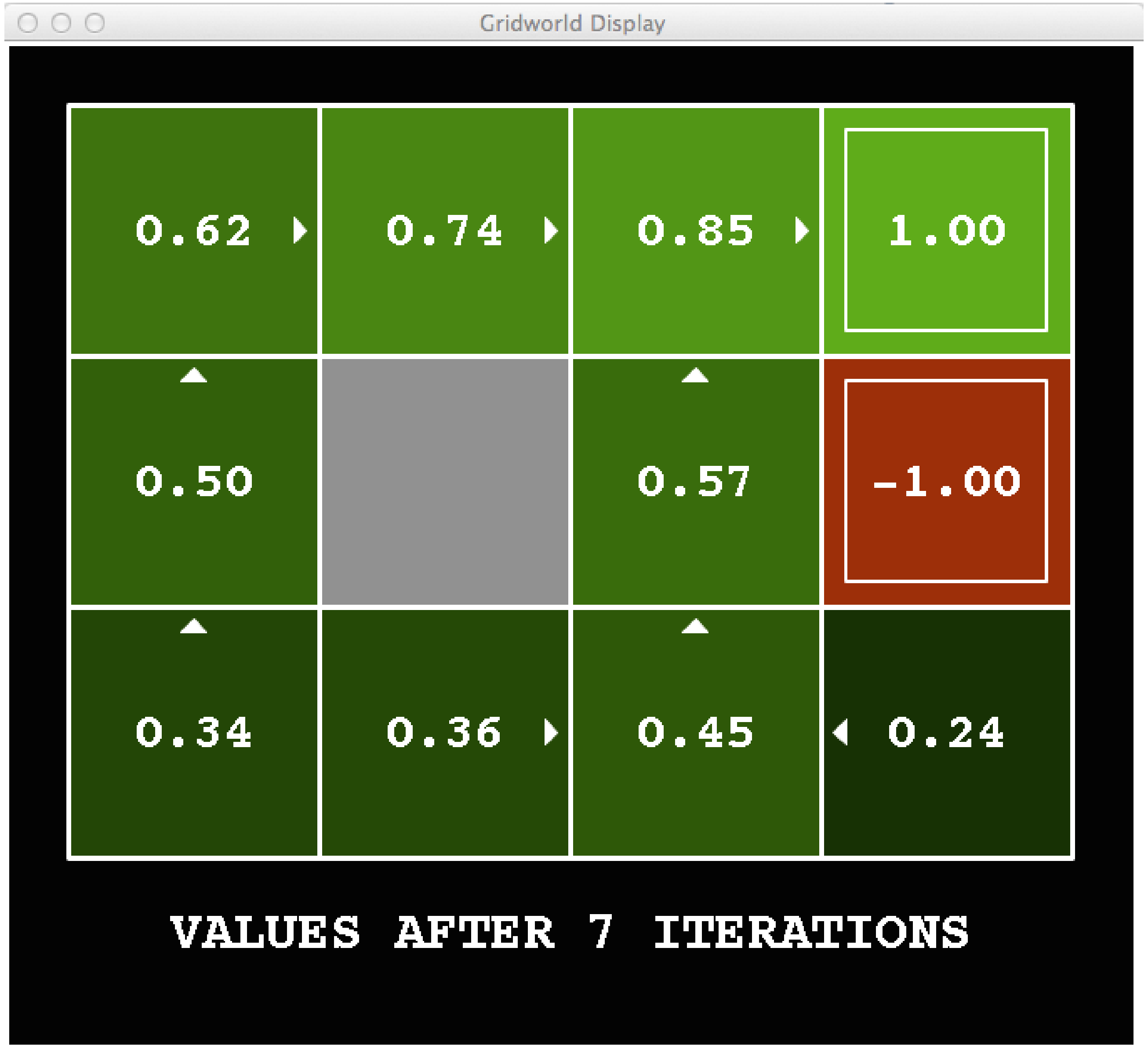
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



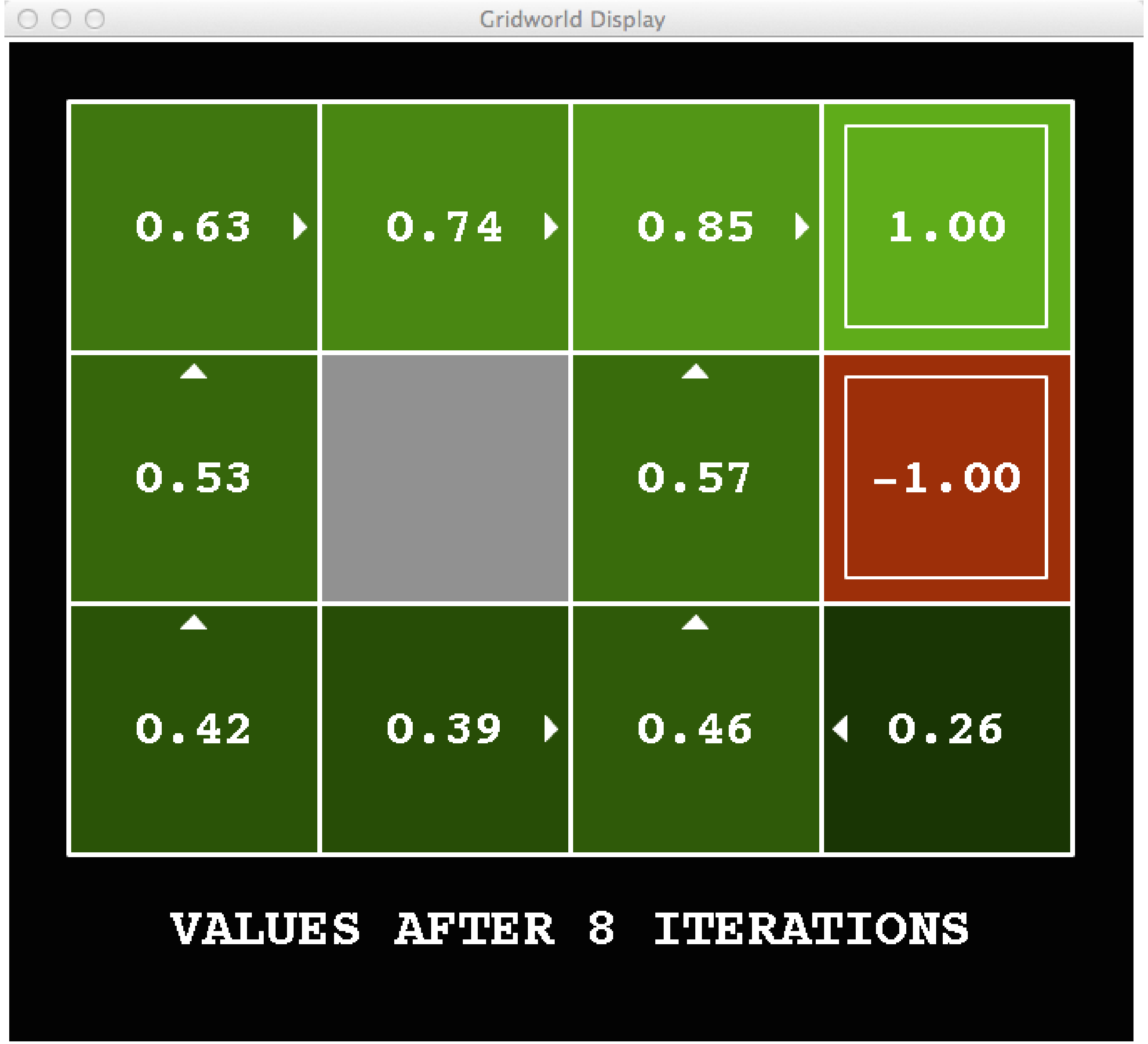
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



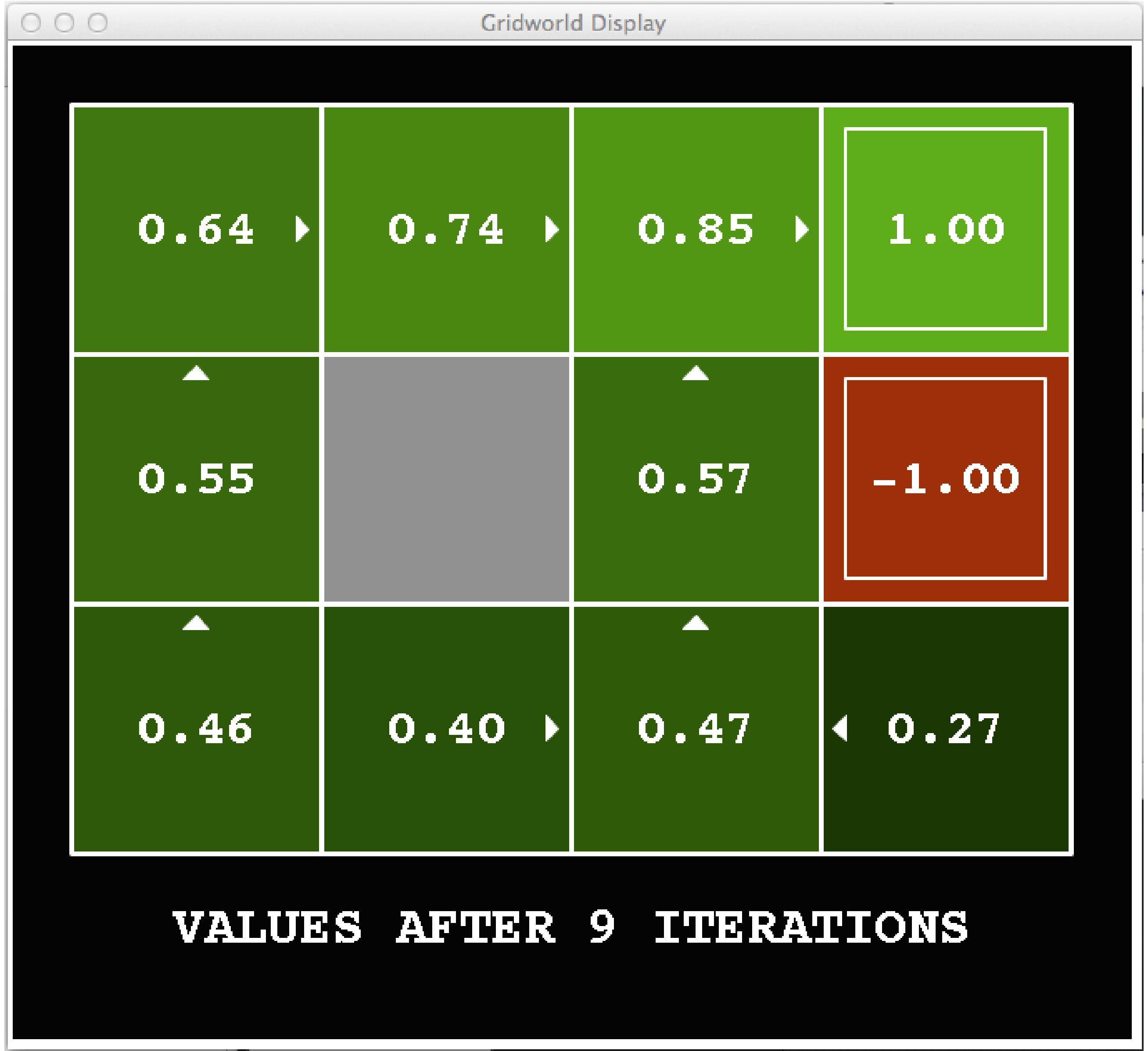
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



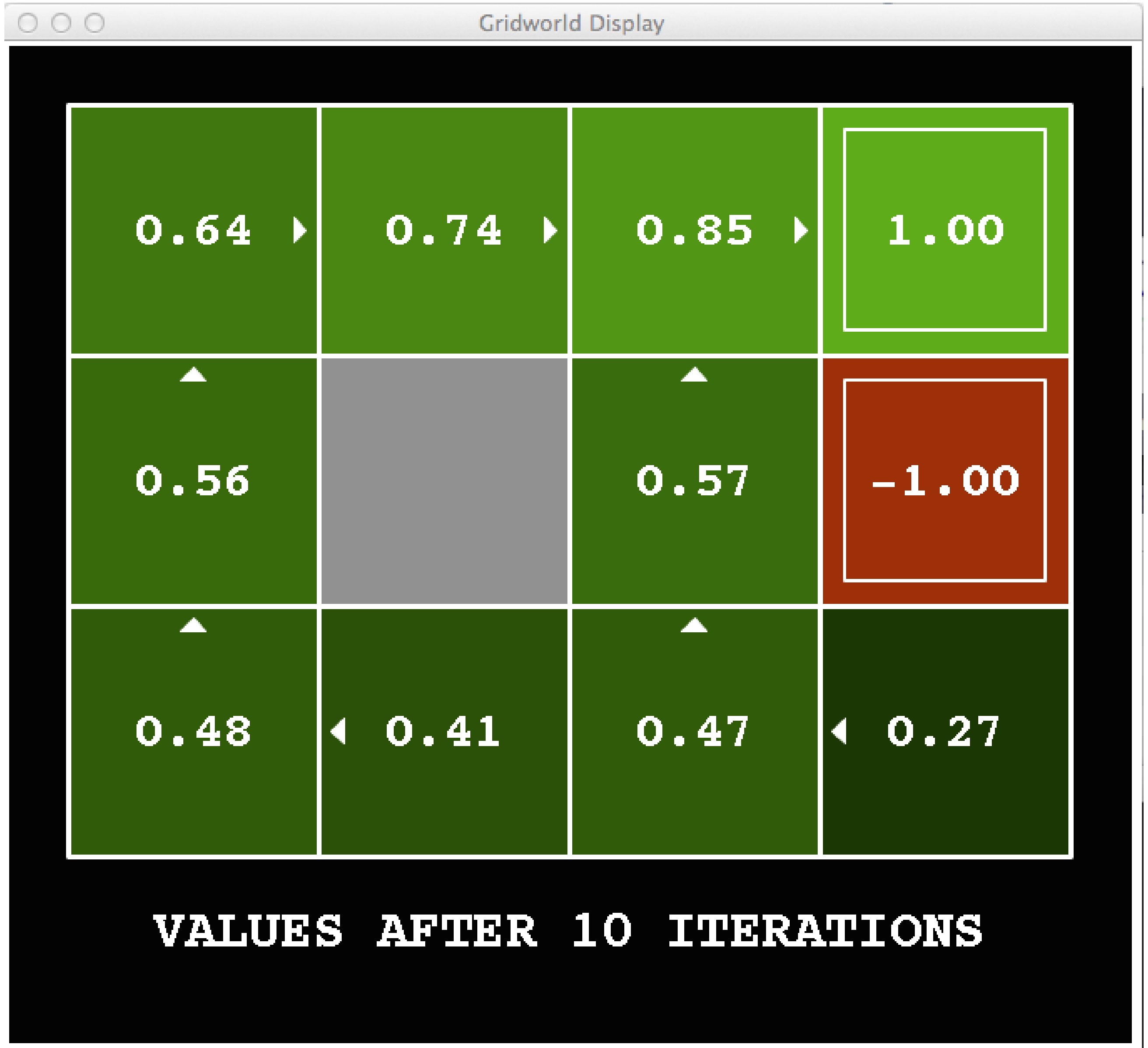
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



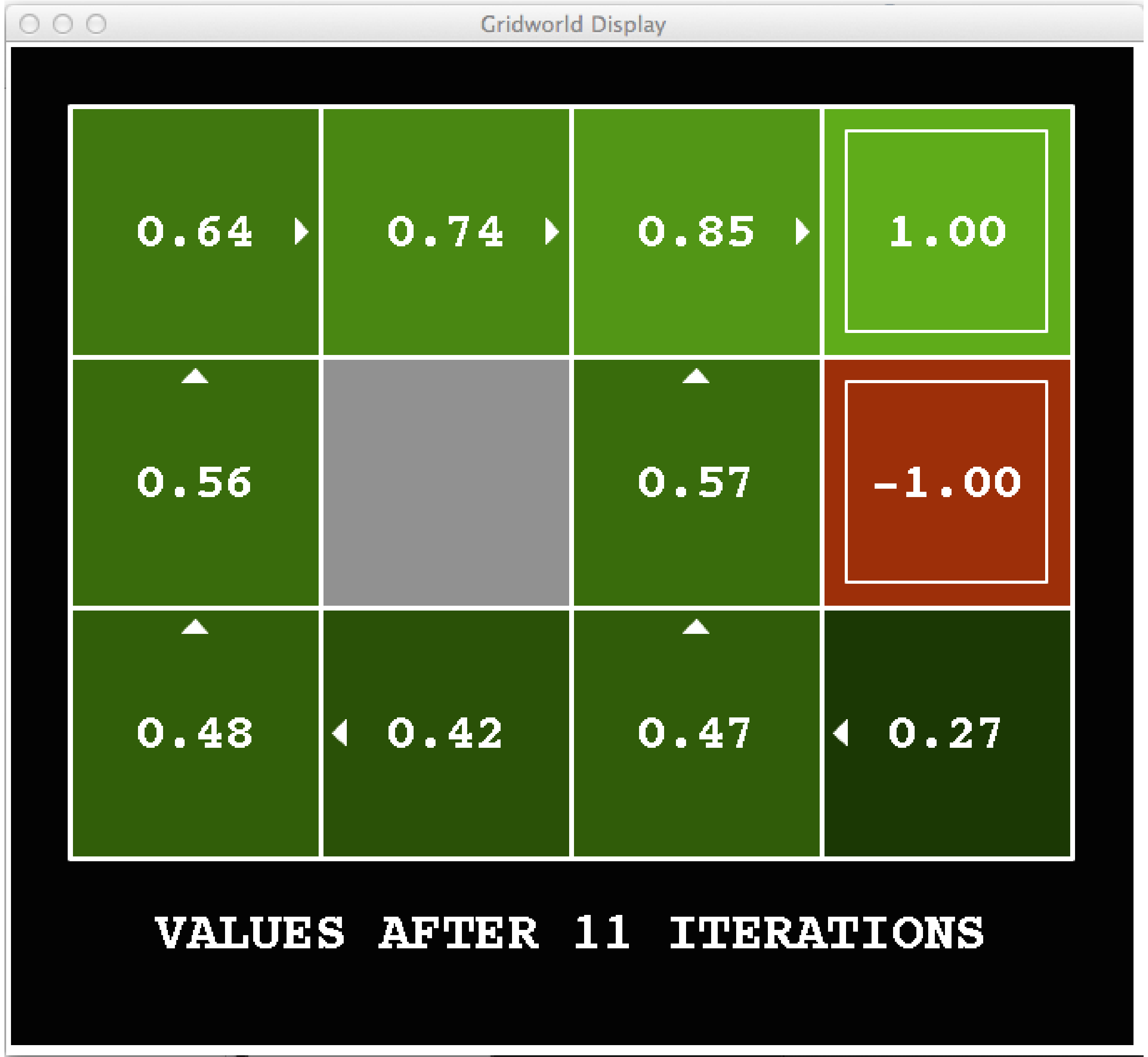
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



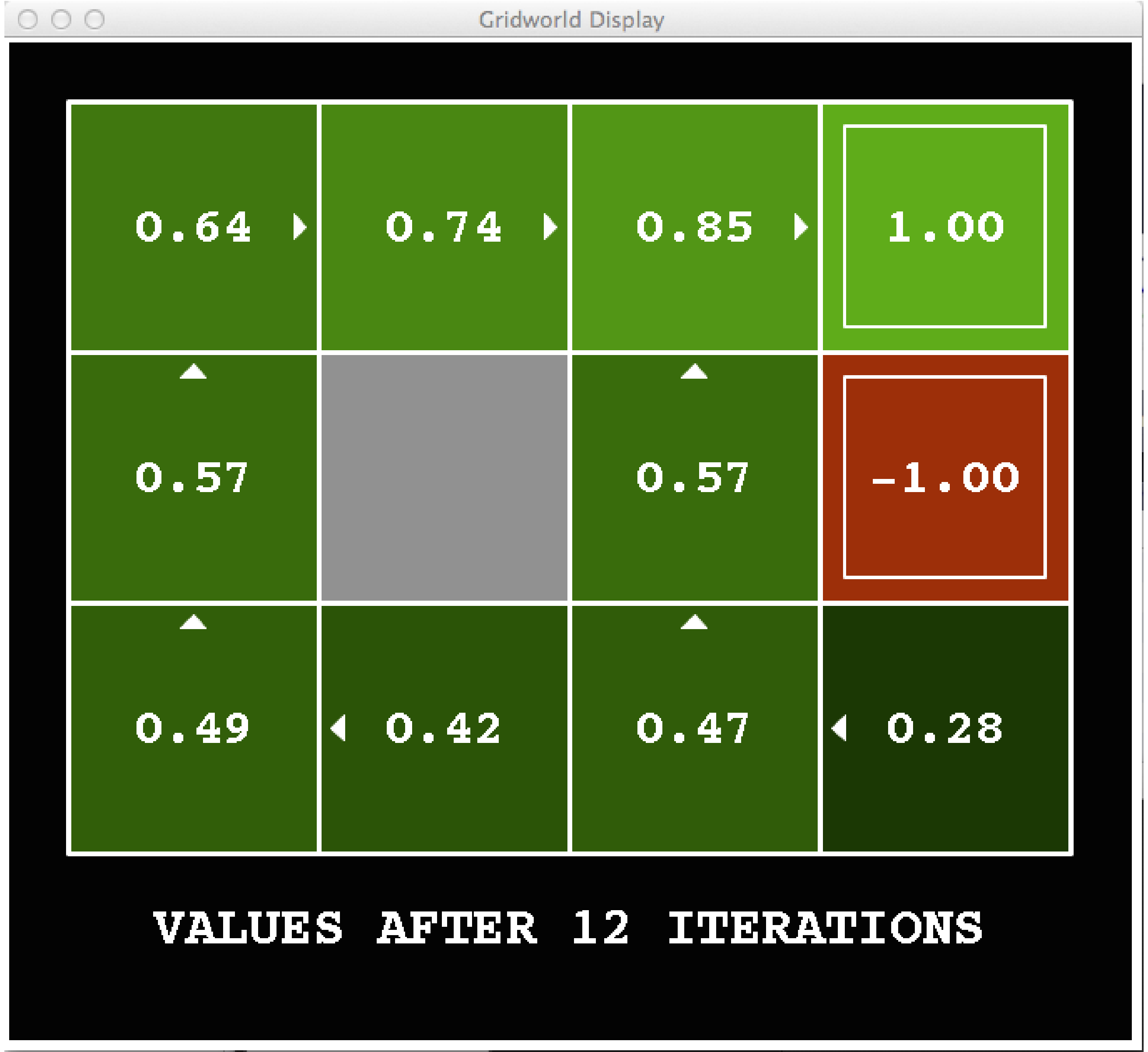
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



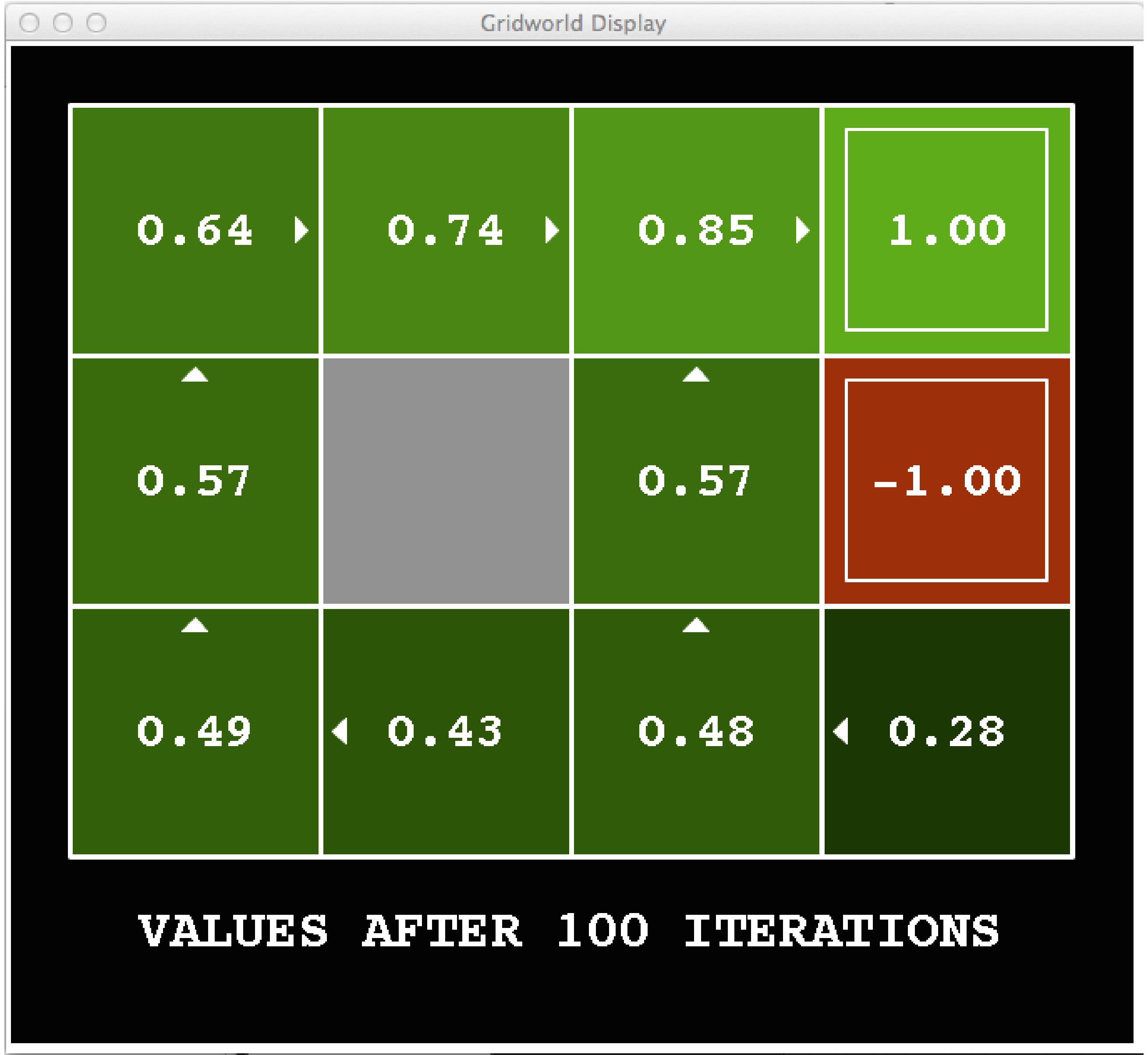
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

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

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

Policy Iteration

- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

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

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s')\right]$$

Policy iteration algorithm

```
function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states S, actions A(s), transition model P(s′ | s, a)
  local variables: U, a vector of utilities for states in S, initially zero
                   π, a policy vector indexed by state, initially random

  repeat
      U ← POLICY-EVALUATION(π, U, mdp)
      unchanged? ← true
      for each state s in S do
          a* ← argmax_{a∈A(s)} Q-VALUE(mdp, s, a, U)
          if Q-VALUE(mdp, s, a*, U) > Q-VALUE(mdp, s, π[s], U) then
              π[s] ← a*; unchanged? ← false
  until unchanged?
  return π
```

Figure 16.9 The policy iteration algorithm for calculating an optimal policy.

Comparison

- Both value iteration and policy iteration compute the same thing (all optimal values)

- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don’t track the policy, but taking the max over actions implicitly recomputes it

- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we’re done)

- Both are dynamic programs for solving MDPs

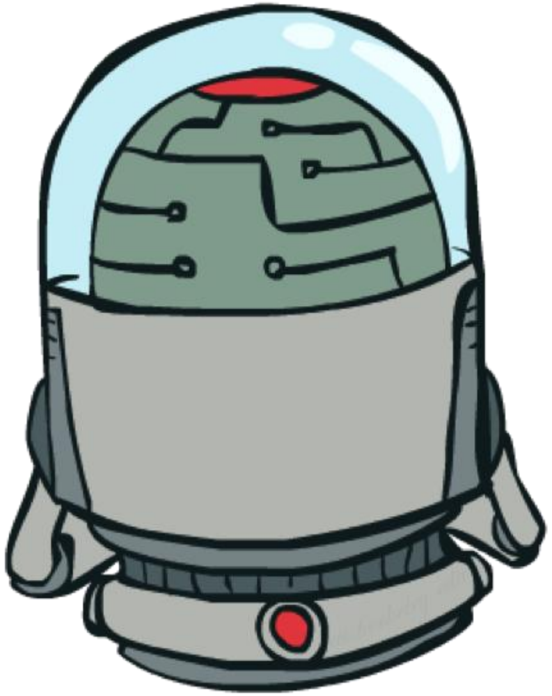
Summary: MDP Algorithms

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

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

Double Bandits

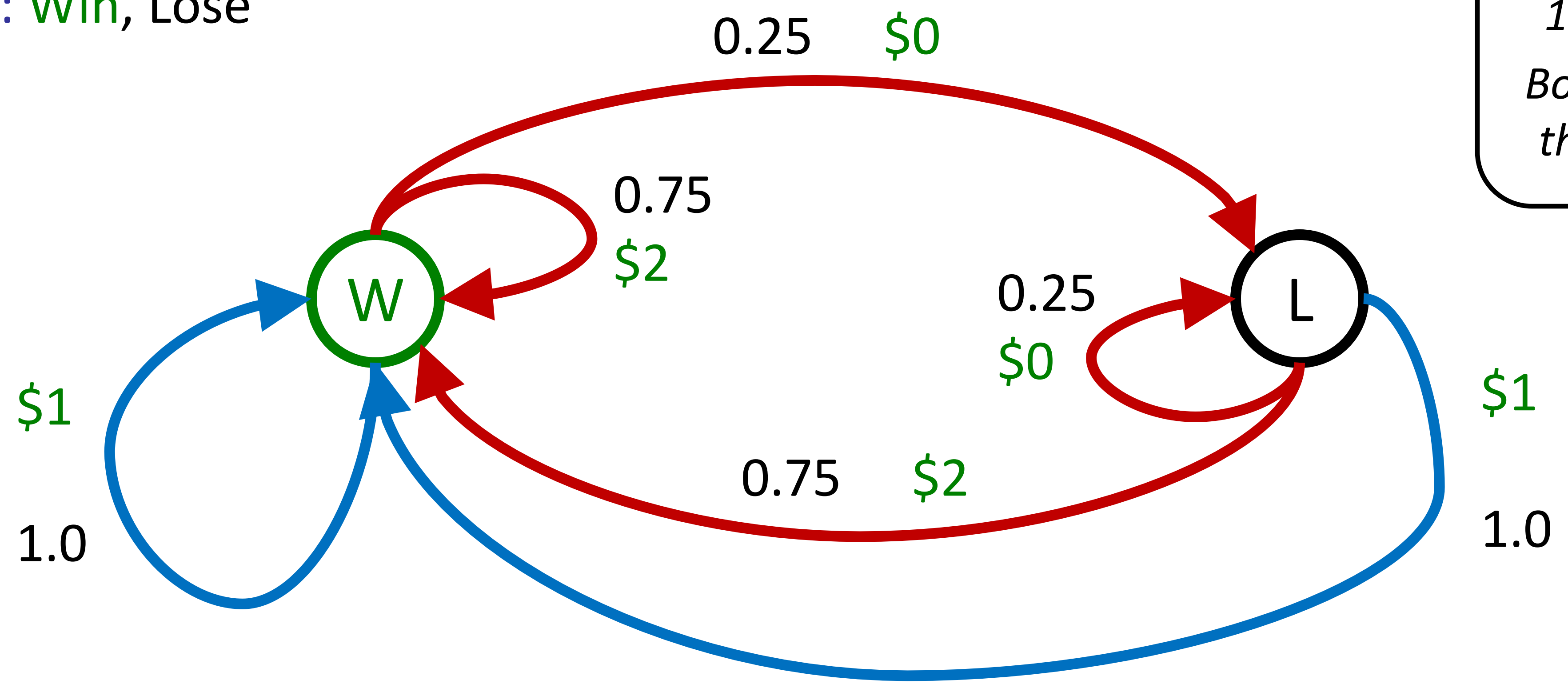
Blue slot machine gives you $1 when you pull the lever



Red slot machine gives you $0 or $2 when you pull the lever

Double-Bandit MDP

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



No discount

100 time steps

Both states have the same value

Offline Planning

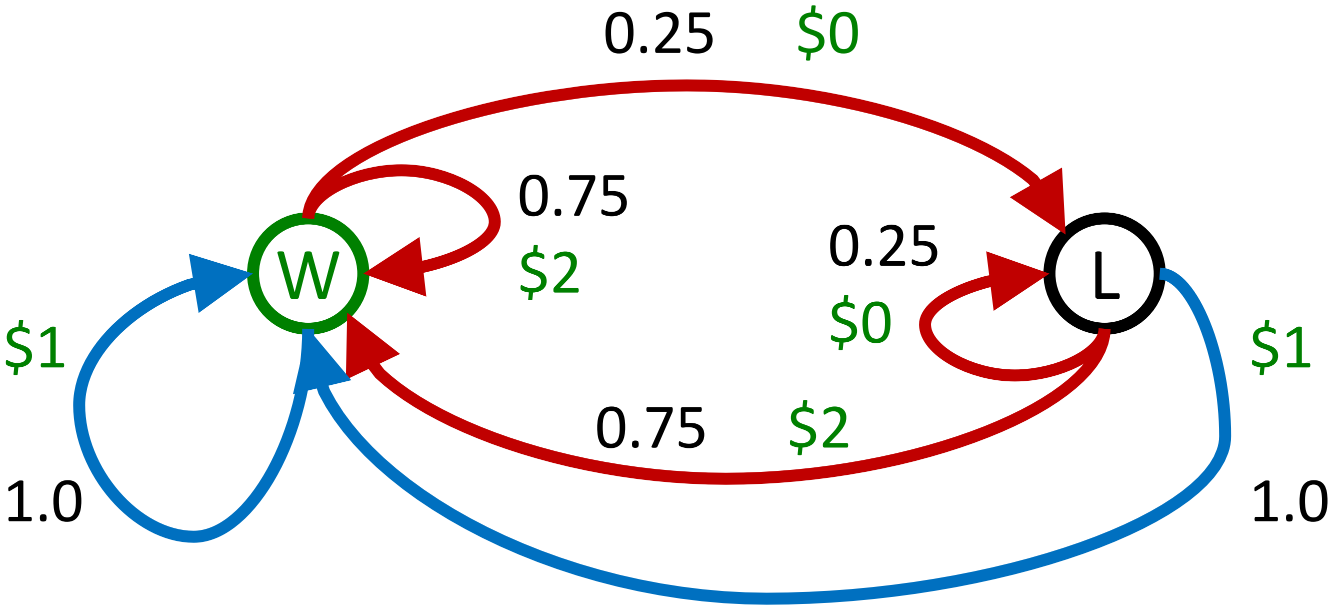
- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

No discount

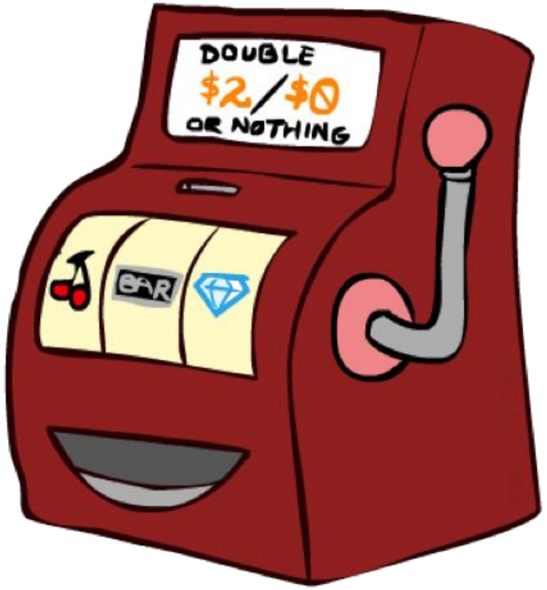
100 time steps

Both states have the same value

	Value
Play Red	150
Play Blue	100



Let’s Play!

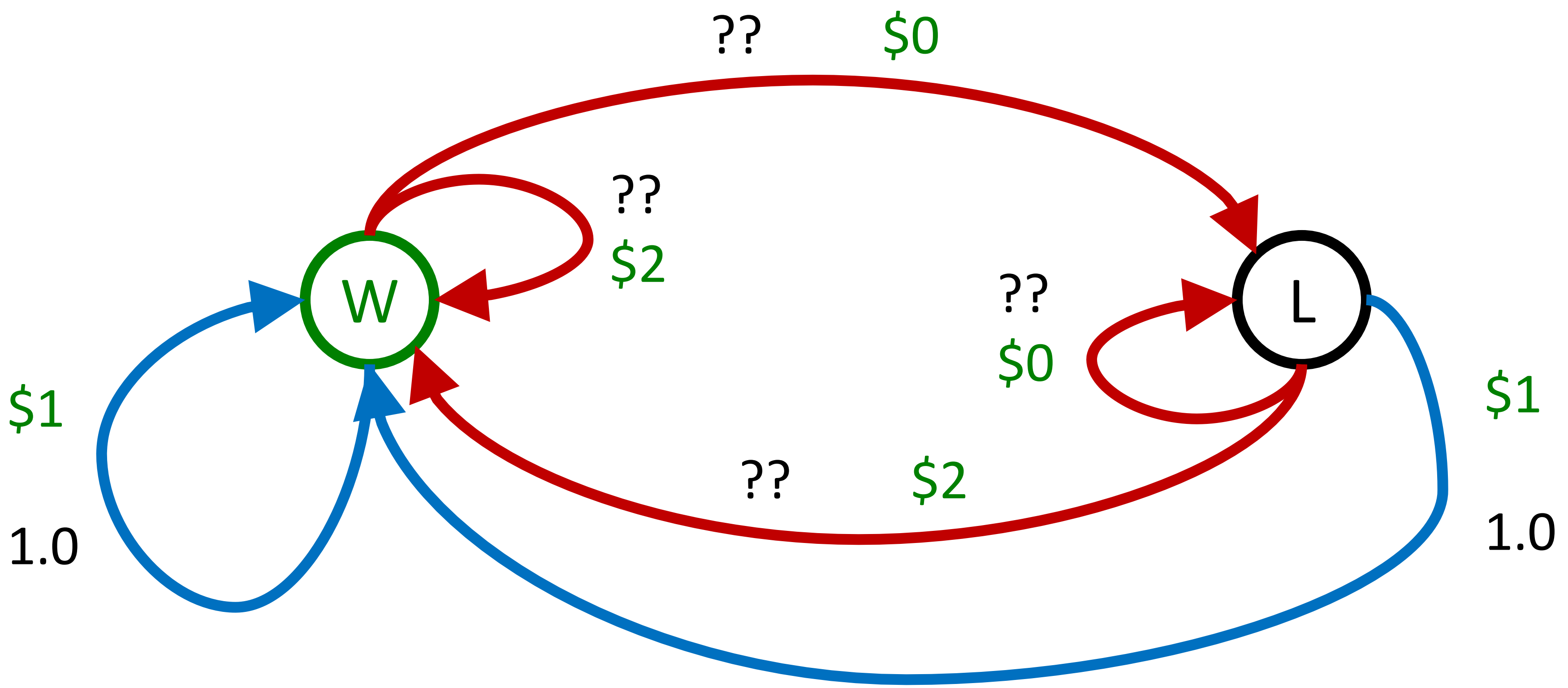


$2 $2 $0 $2 $2

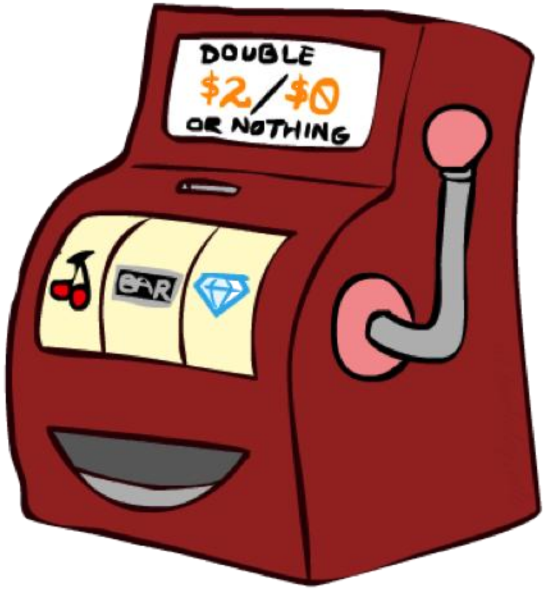
$2 $2 $0 $0 $0

Online Planning

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



Let's Play!



$0 $0 $0 $2 $0

$2 $0 $0 $0 $0

What Just Happened?

- That wasn't planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn't solve it with just computation
  - You needed to actually act to figure it out
- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP

Next Time: Reinforcement Learning!