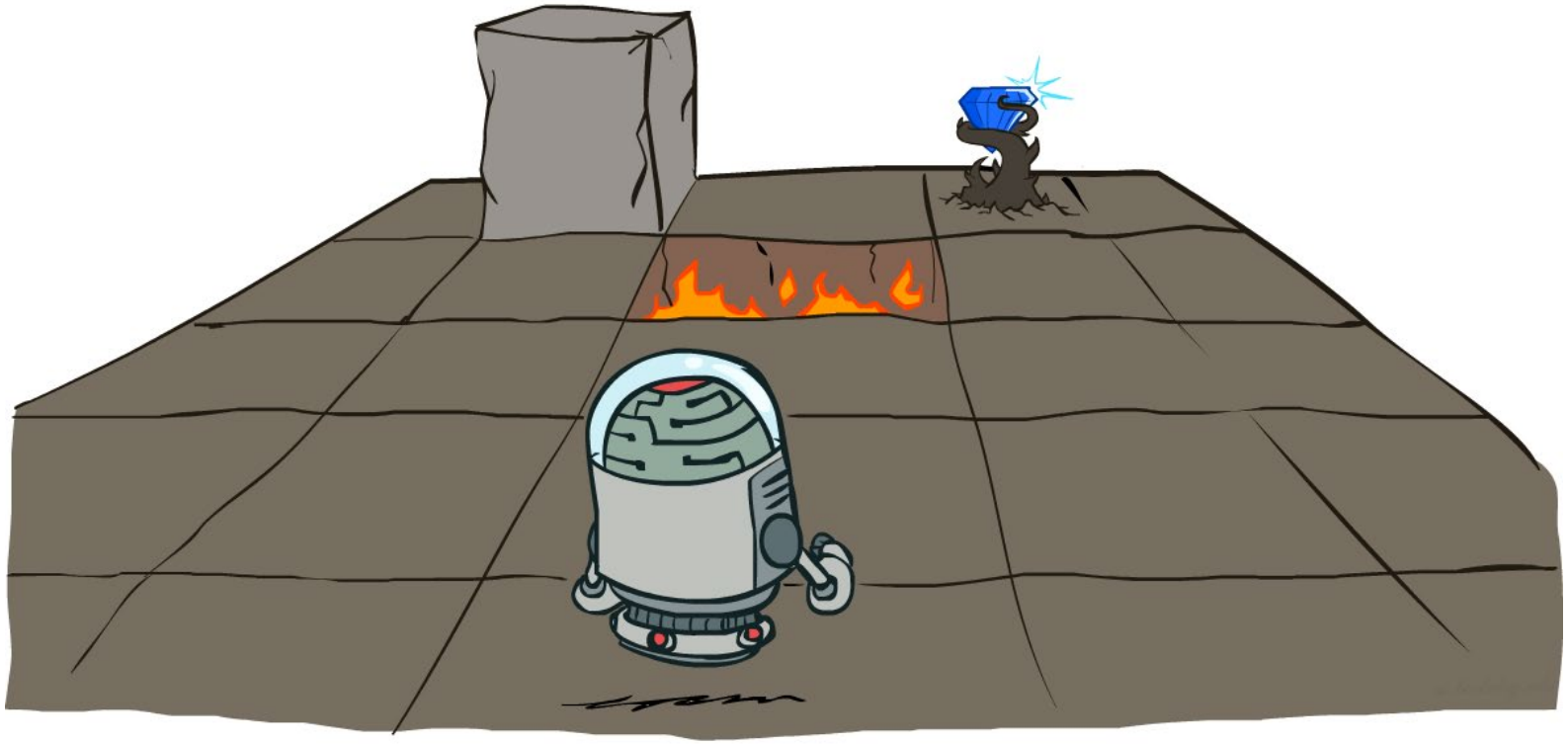
Markov Decision Processes

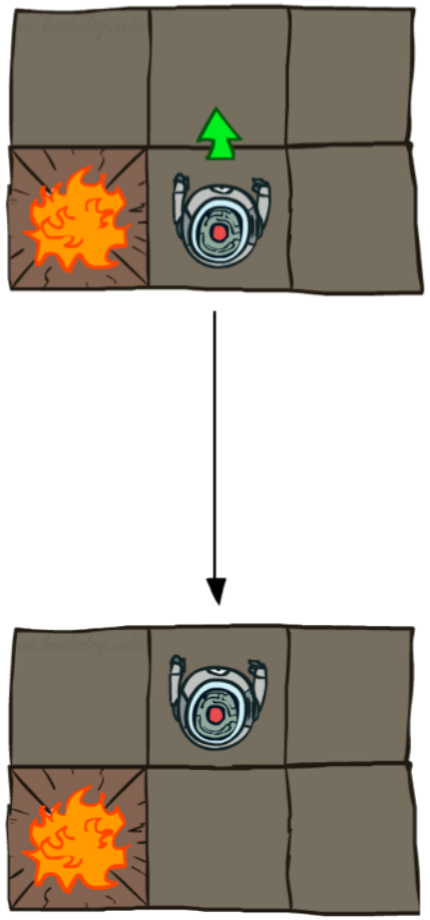


Outline

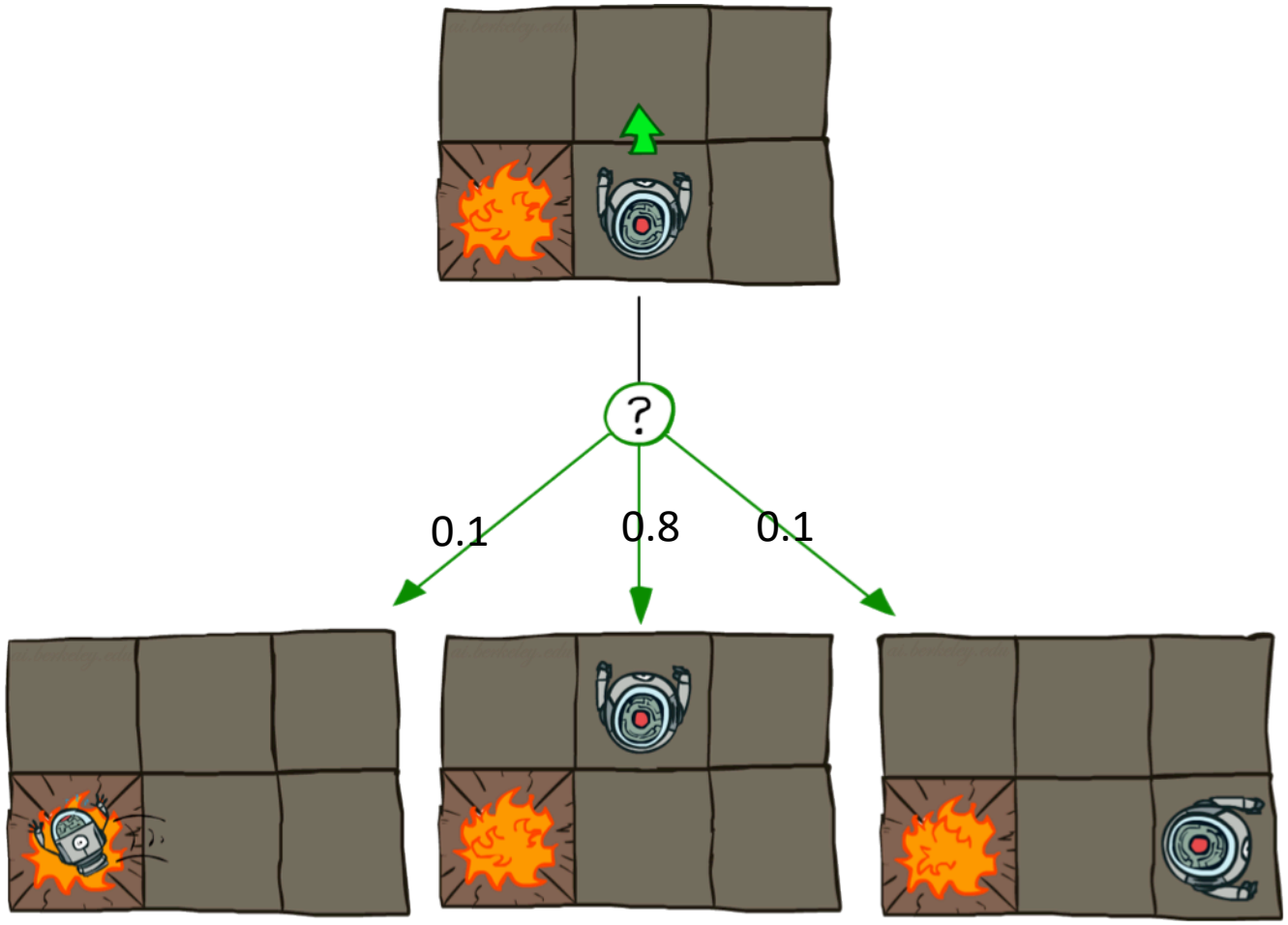
- MDP
- Solving MDP
  - Value Iteration
  - Policy Iteration

Non-Deterministic Search

Deterministic Grid World

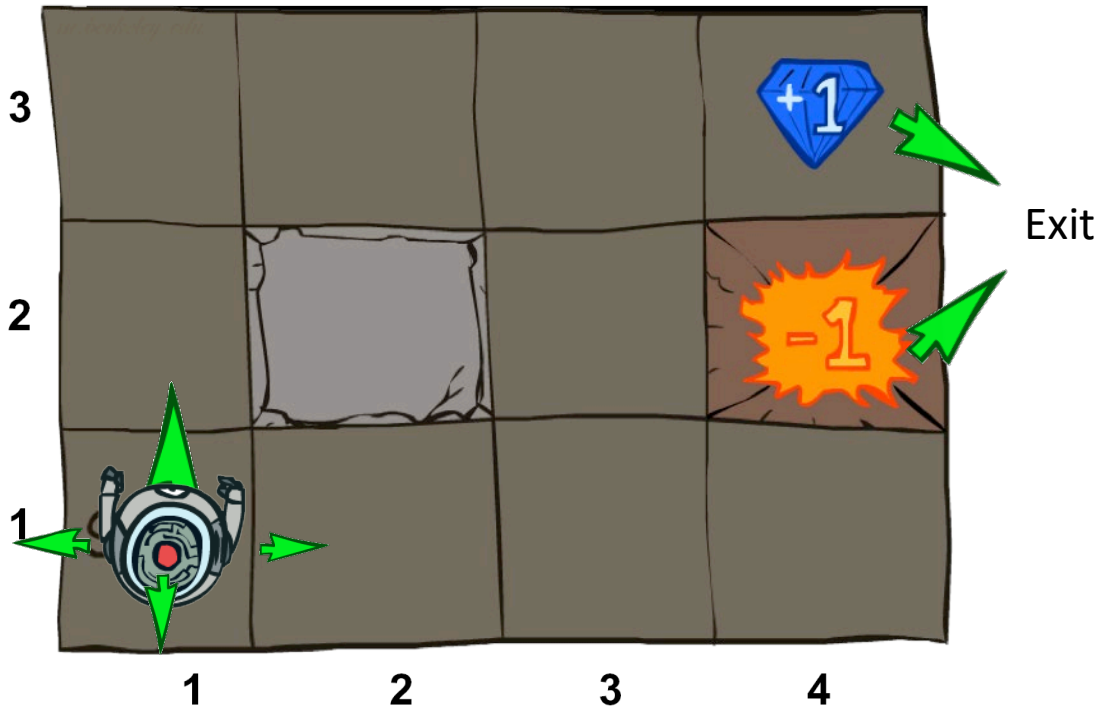


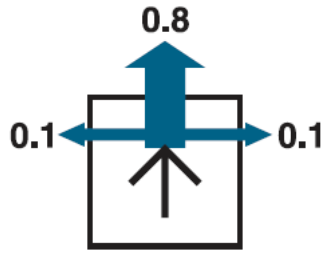
Stochastic Grid World



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 in the same square
- The agent receives rewards each time step
  - Small "living" reward for nonterminal states (can be negative, -0.04)
  - Big rewards come for terminal states (good +1 or bad -1)
- Goal: maximize sum of rewards

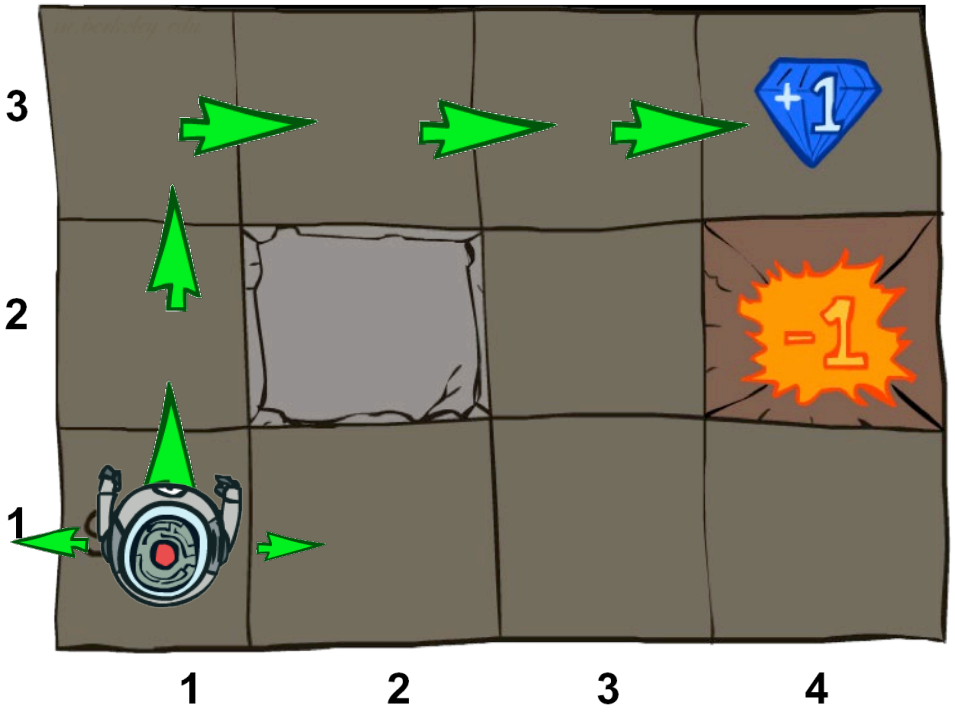


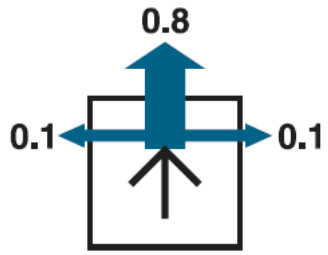


Probabilities of Reaching the Goal

From (1,1) by the action sequence [Up, Up, Right , Right , Right], the probability of reaching the goal state at (4, 3) is $0.8^5 = 0.32768$.

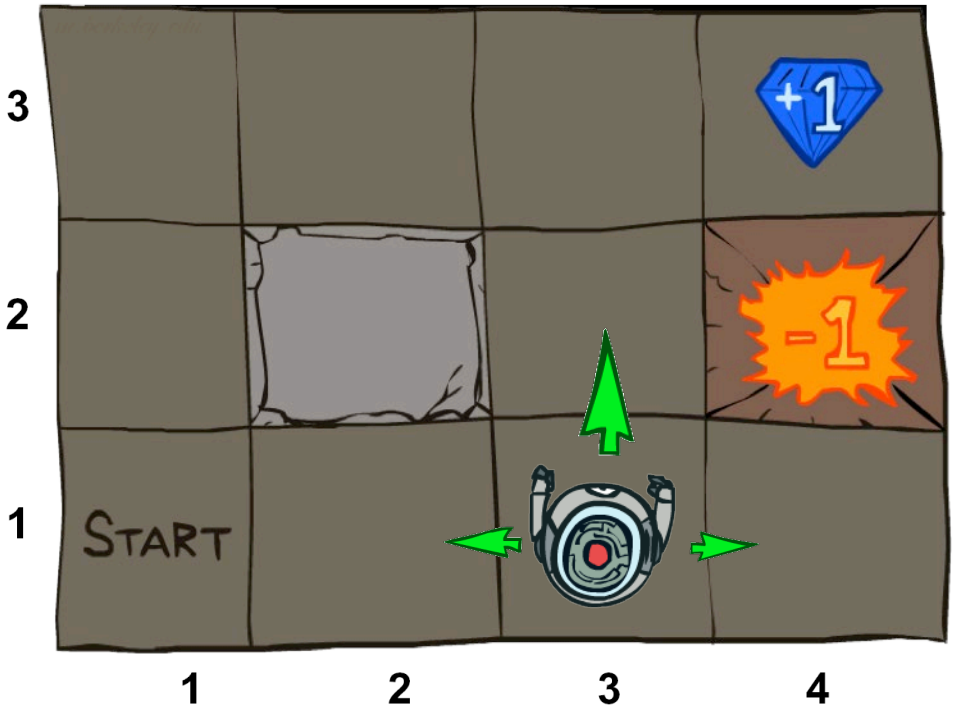
There is also a small chance of accidentally reaching the goal by going the other way around with probability $0.1^4 * 0.8 = 0.32776$.





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., $P(s' \mid s, a)$
      - $P((3,2) \mid (3, 1), up) = 0.8$
      - $P((2,1) \mid (3, 1), up) = 0.1$
      - $P((4,1) \mid (3, 1), up) = 0.1$
    - 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



Video of Demo Gridworld Manual Intro

What is Markov about MDPs?

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

- For Markov decision processes, "Markovian" 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)

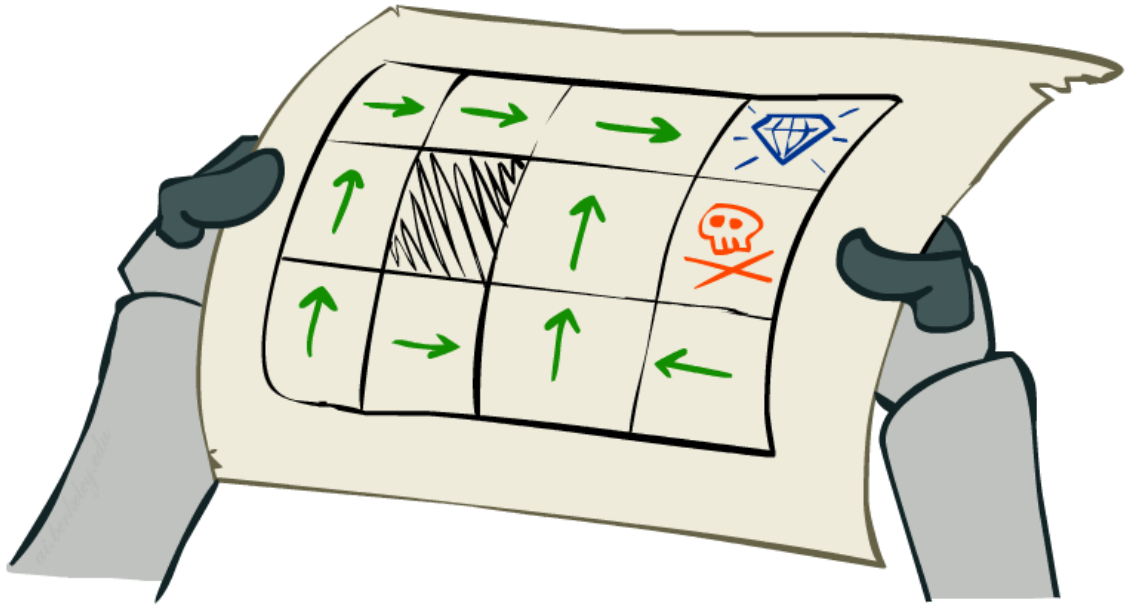
Policies

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

- For MDPs, we want an optimal

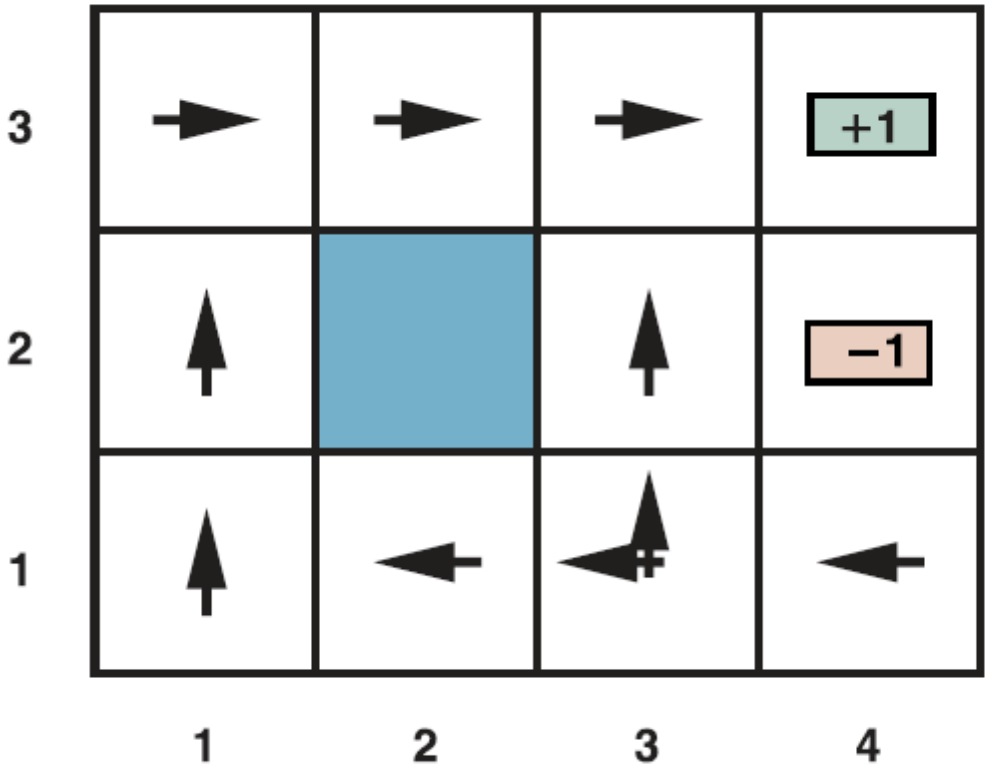
  policy $\pi^*: S \rightarrow A$

  - A policy π gives an action for ***each*** state
  - An optimal policy is one that maximizes ***expected utility*** if followed



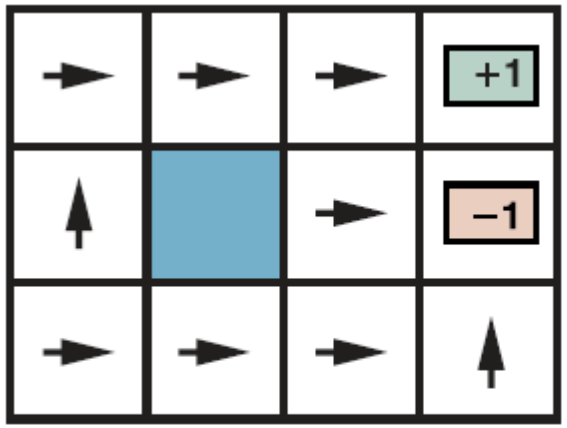
Optimal policy for all nonterminal s

Optimal Policies



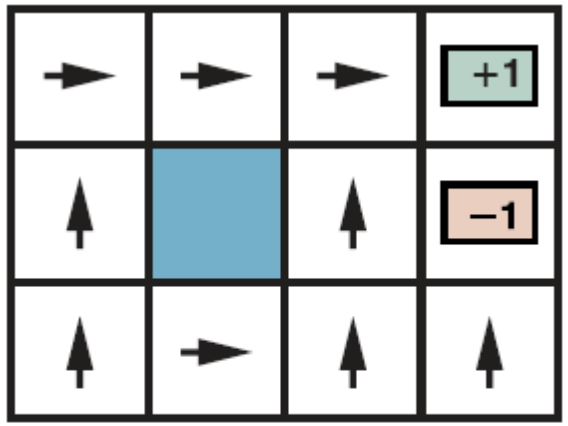
r = -0.04

Life is painful



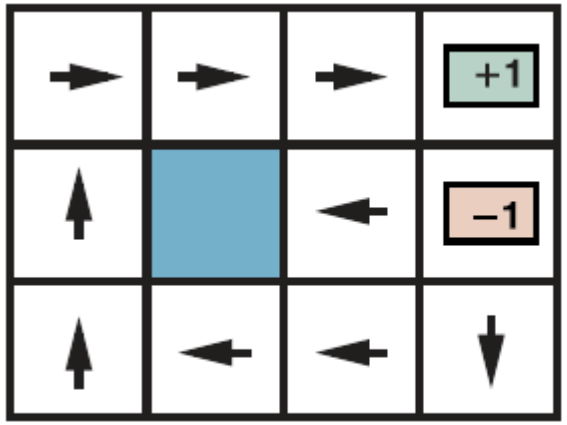
$r < -1.6497$

Life is unpleasant



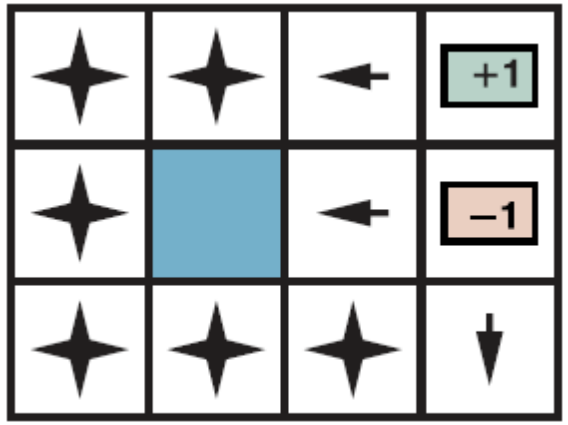
$-0.7311 < r < -0.4526$

Life is slightly dreary



$-0.0274 < r < 0$

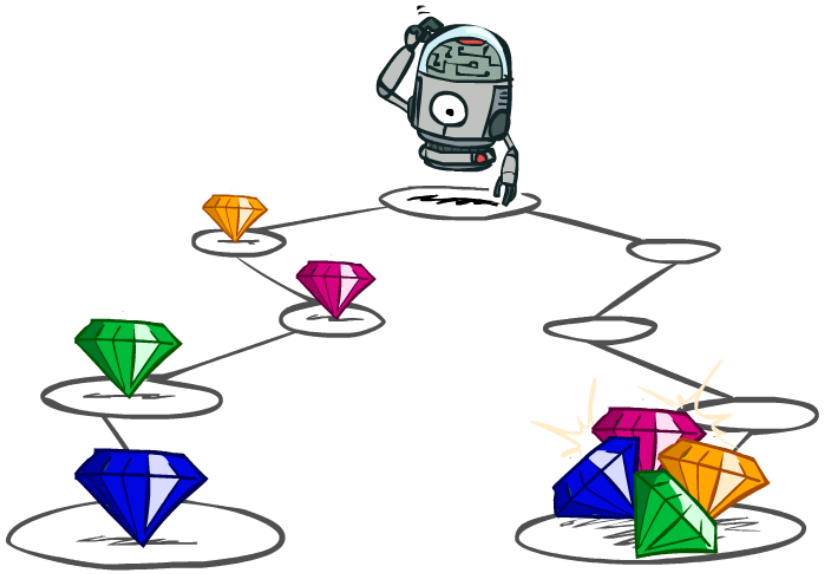
Life is enjoyable



$r > 0$

Utilities of State 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

1

Worth Now

γ

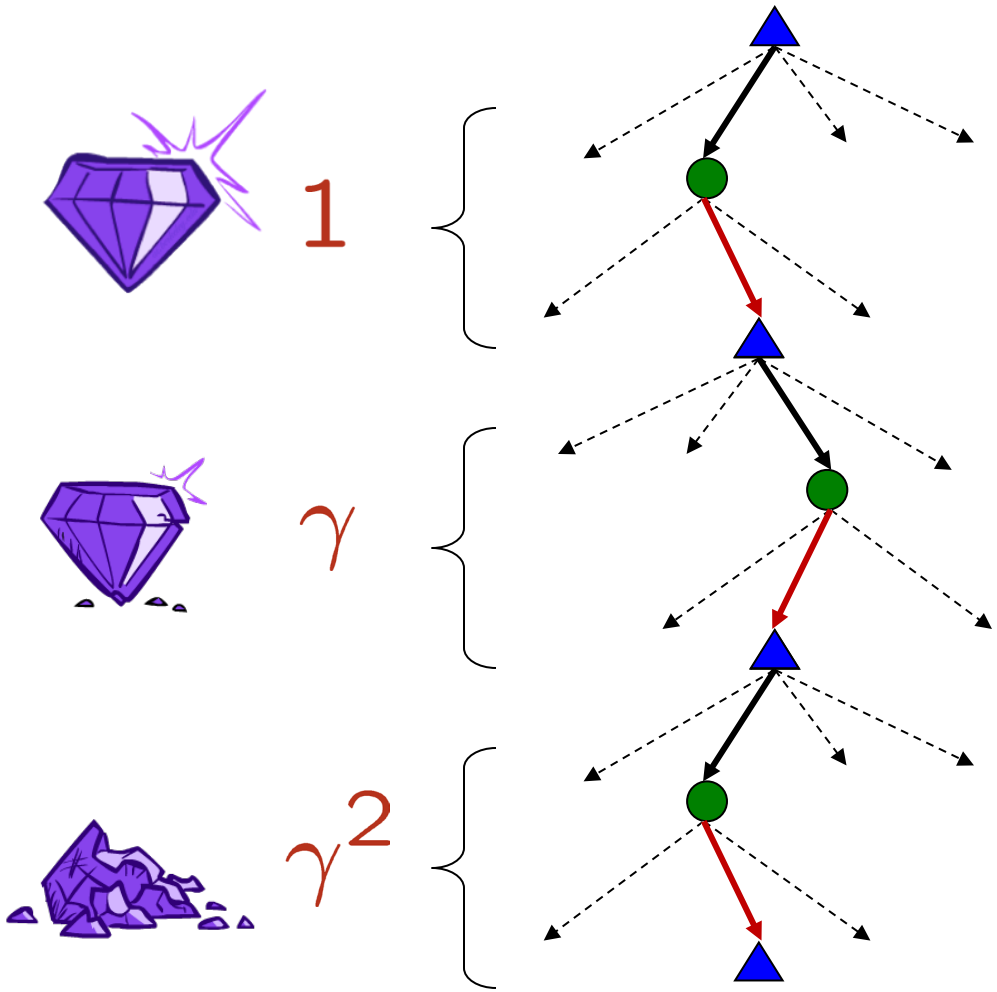
Worth Next Step

γ^2

Worth In Two Steps

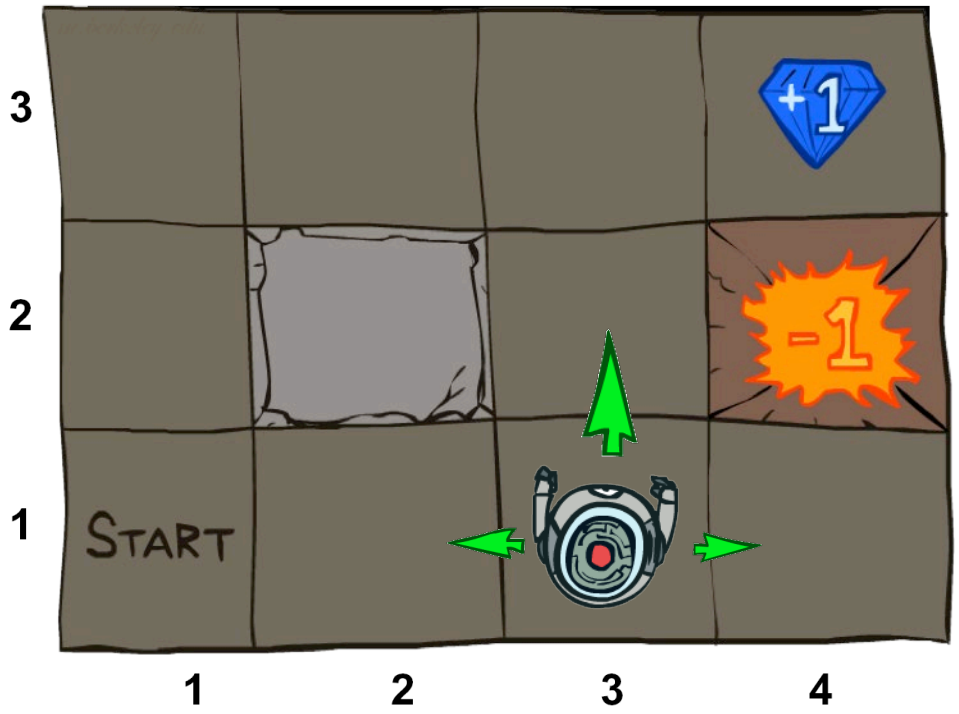
Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- 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])
- Why discount?
  - Simulate human nature.
  - In Economic, Υ is equivalent to an interest rate of $\frac{1}{\Upsilon} - 1$.
  - Helps our algorithms converge.



Stationary Preferences

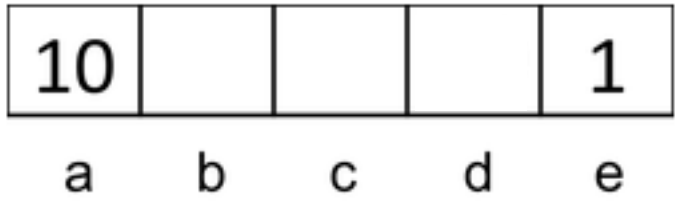
- A policy that depends on the time is called nonstationary.
  - For example, if an agent starts at (3,1) and is given 3 steps, the optimal action is to go UP. But this will risk to hit the -1 state.
  - If the agent is given 100 steps, there is plenty time to take the safe route by going LEFT.
  - With a finite time, an optimal action depends on how much time is left.



  - With an infinite time, there is no reason to behave differently in the same state at different times. In this case, the optimal policy is stationary.

IcP: Discounting

- Given:

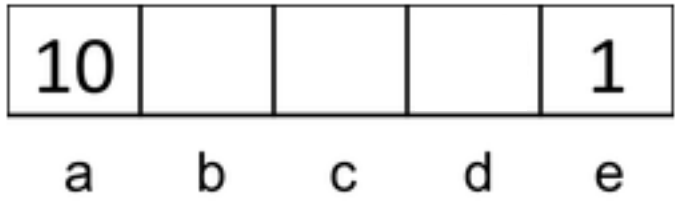


  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

- Q1: For $\gamma = 1$, what is the optimal policy?
- Q2: For $\gamma = 0.1$, what is the optimal policy?
- Q3: For which γ are West and East equally good when in state d?

Solution

- Given:



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

- Q1: For $\gamma = 1$, what is the optimal policy?

10	<-	<-	<-	1

- Q2: For $\gamma = 0.1$, what is the optimal policy?

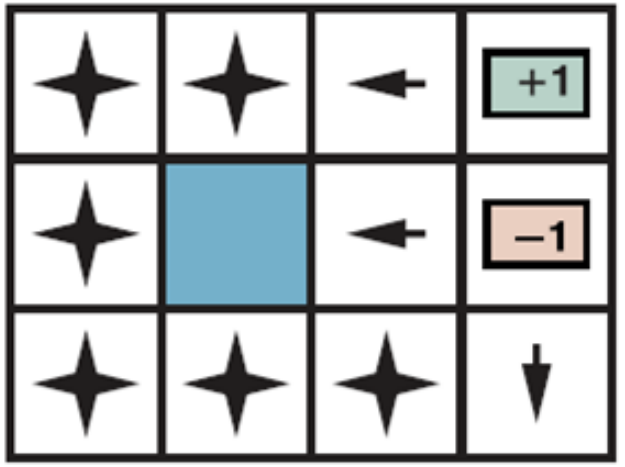
10	<-	<-	->	1

- Q3: For which γ are West and East equally good when in state d?

$$1\gamma = 10\ \gamma^3 \rightarrow \gamma = \frac{1}{\sqrt{10}}$$

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)
    - Problem: gives nonstationary policies (π depends on time left)
  - Discounting: use $0 < \gamma < 1$



$r > 0$

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

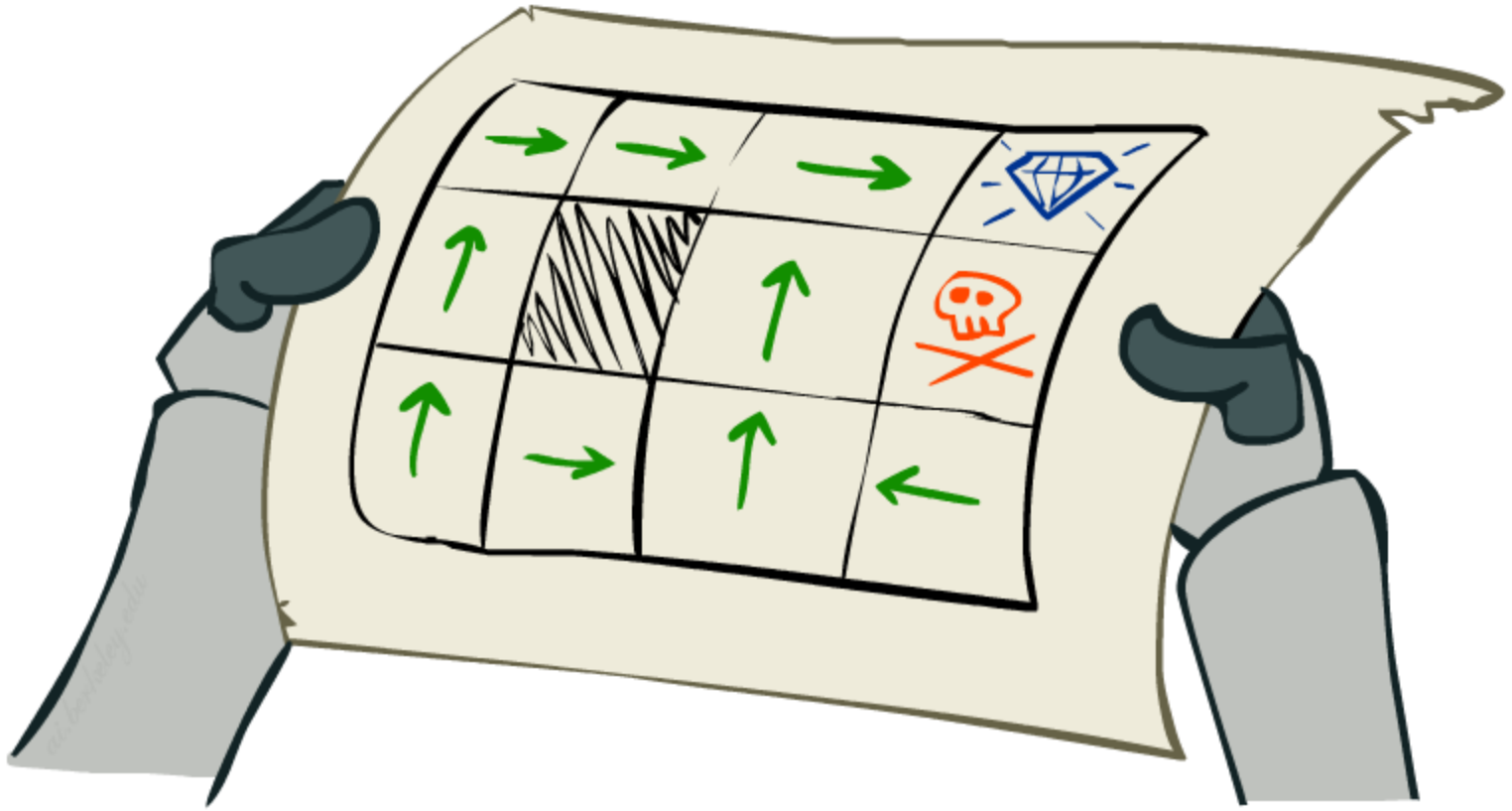
Geometric series

$1+\Upsilon+\Upsilon^2+\Upsilon^3+\ldots = \frac{1}{1-\Upsilon}$

    - Smaller γ means smaller "horizon" – shorter term focus. Larger γ, long-term planning (future rewards still contribute significantly)
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)

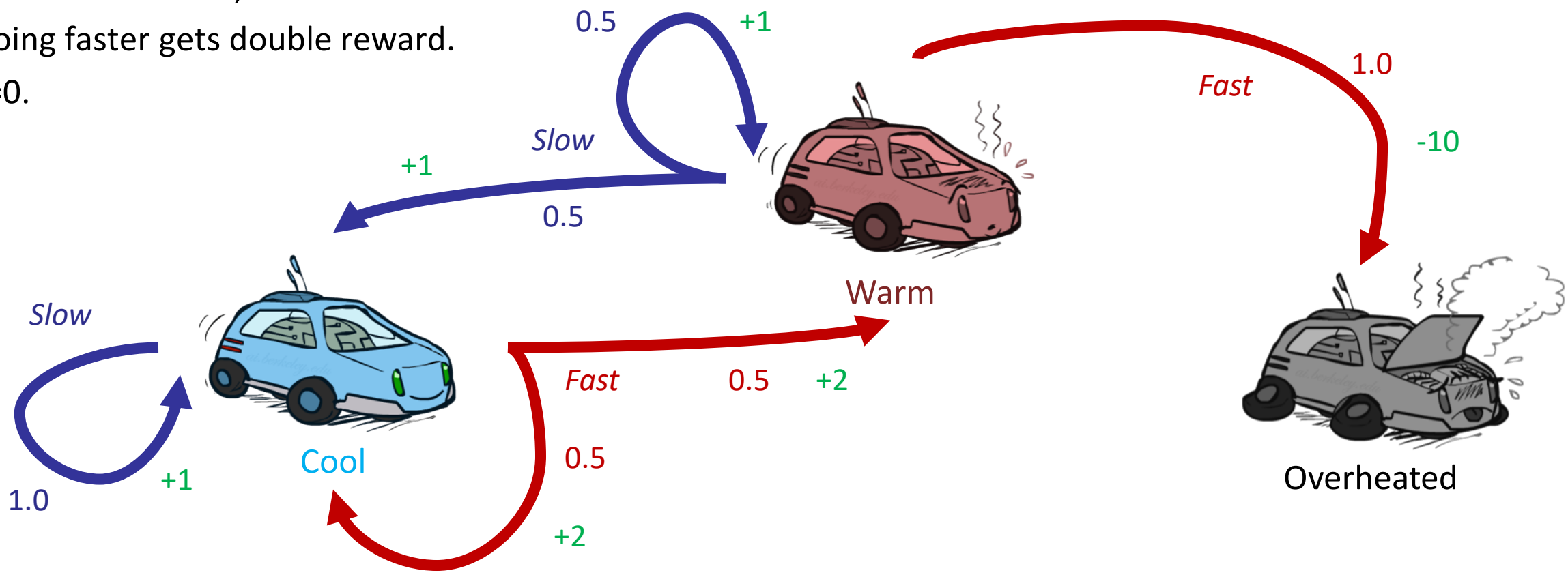
Solving MDPs

- We want to find the optimal policy π^*:
  - Find best action for each state such that it maximizes the expected utility (sum of discounted rewards).
- Methods
  - Value iteration
  - Policy iteration

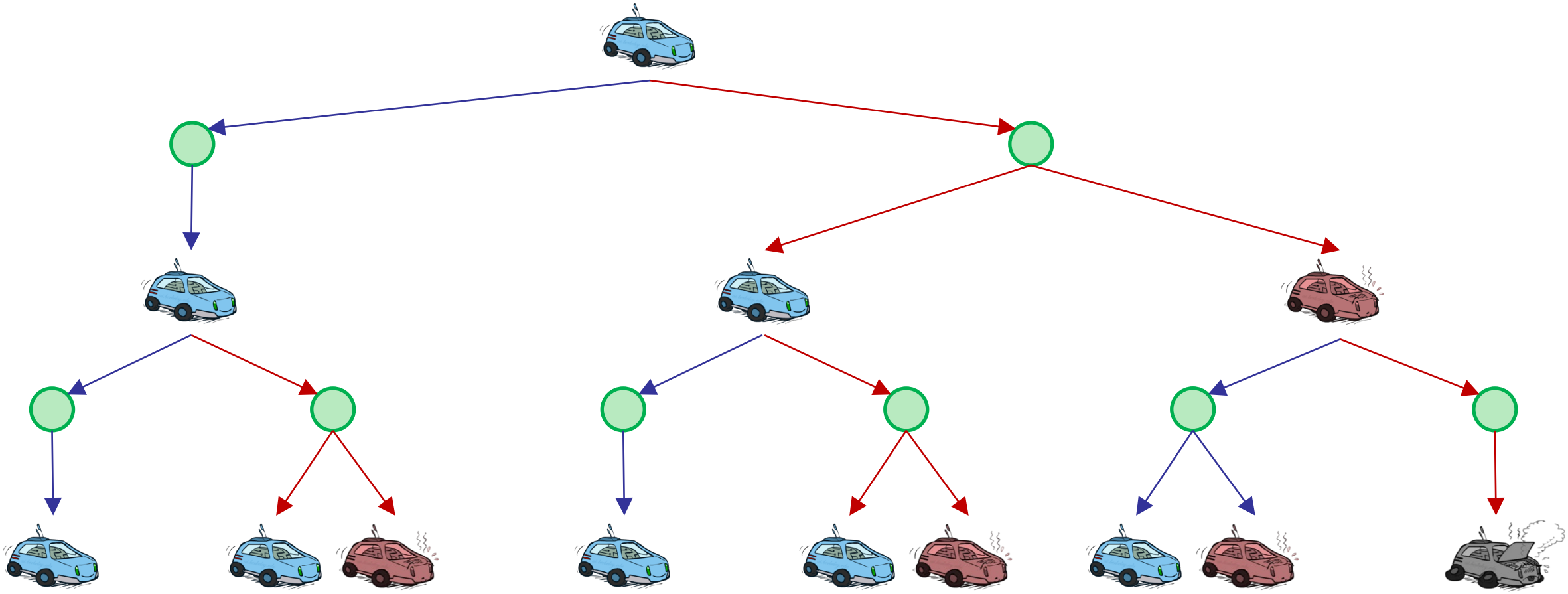


Example: Racing

- A robot car wants to travel far, quickly.
- Three states: Cool, Warm, Overheated.
- Two actions: *Slow*, *Fast.*
- Going faster gets double reward.
- γ=0.

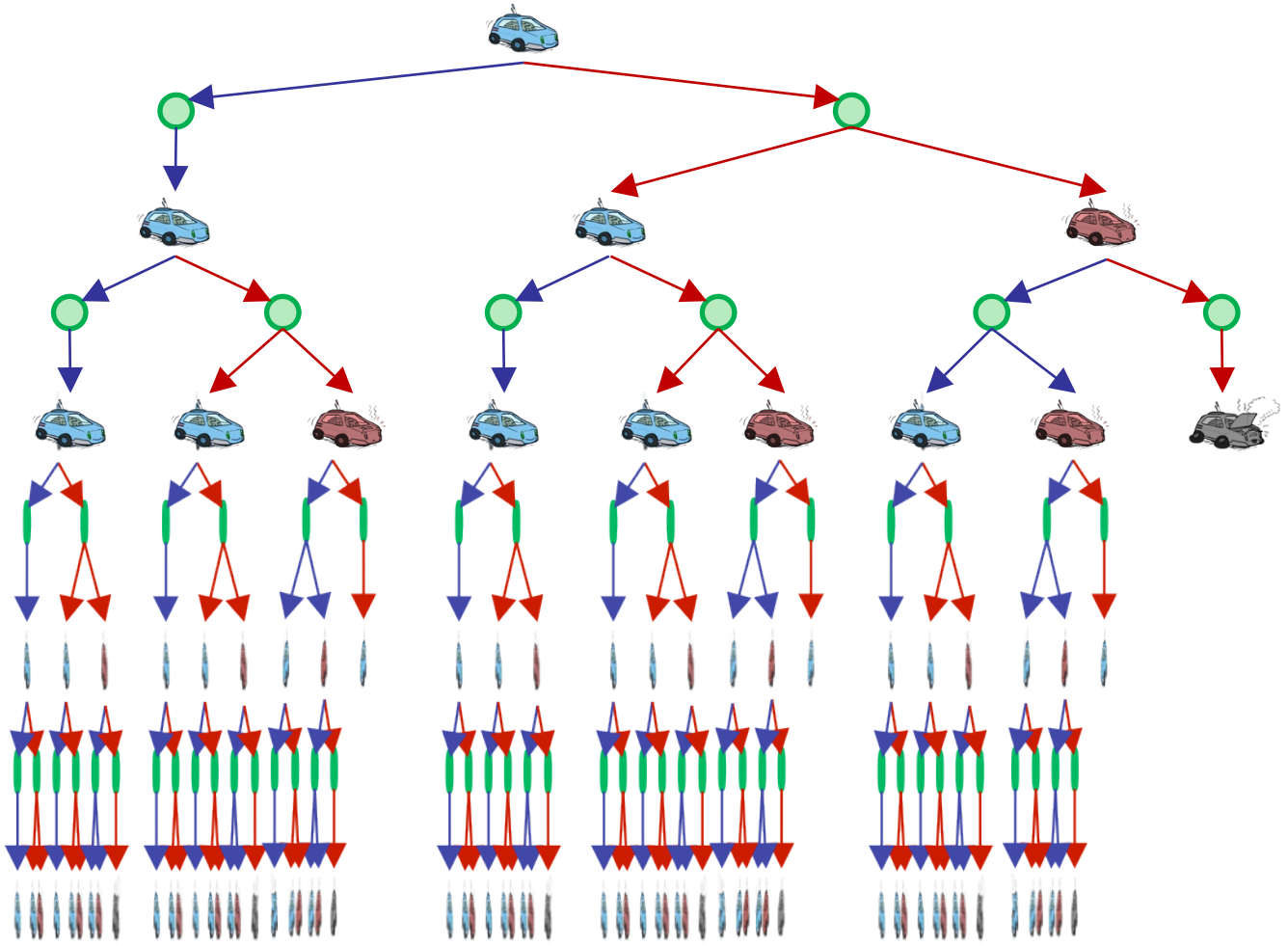


Racing Search Tree

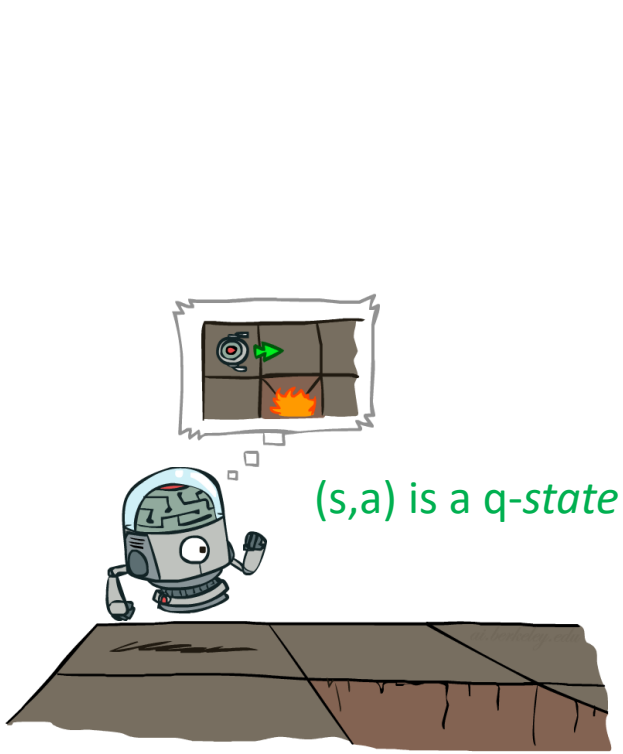


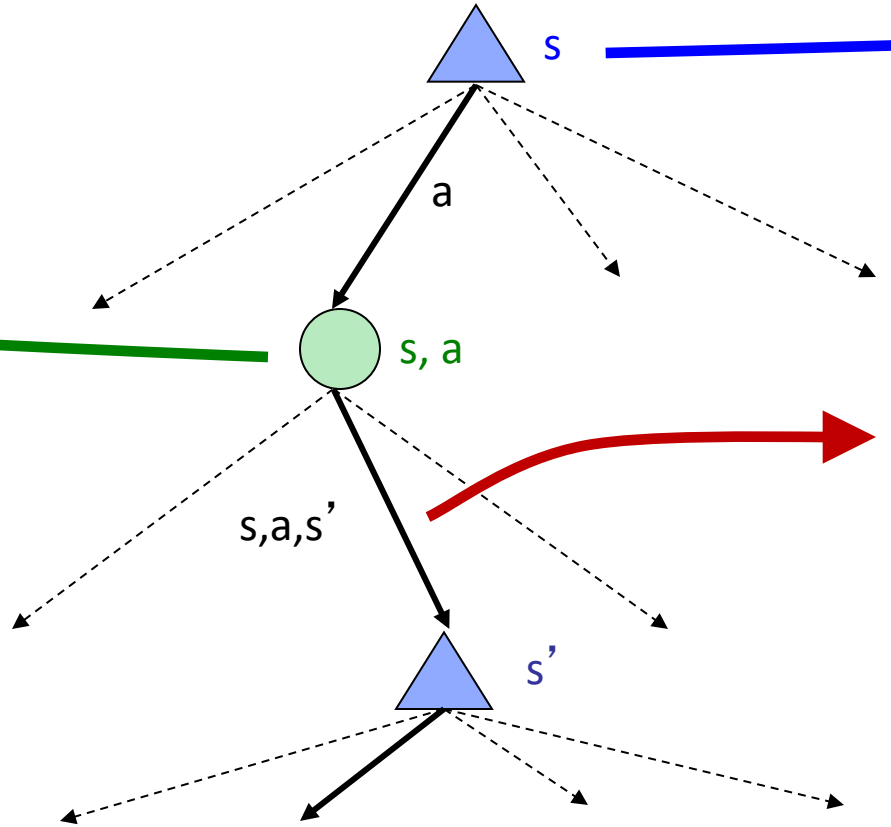
Racing Search Tree

- 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 $\gamma < 1$



MDP Search Trees





s is a *state*

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

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

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

R(s,a,s')

Optimal Quantities

- The value (utility) of a state s:

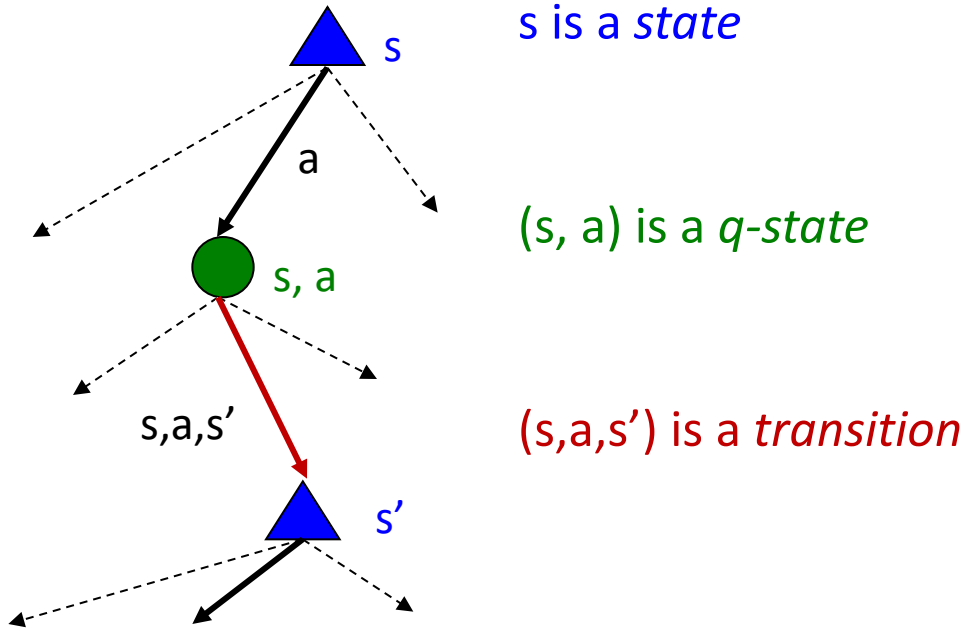
  $V^*(s)$ = expected utility starting in s and acting optimally

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

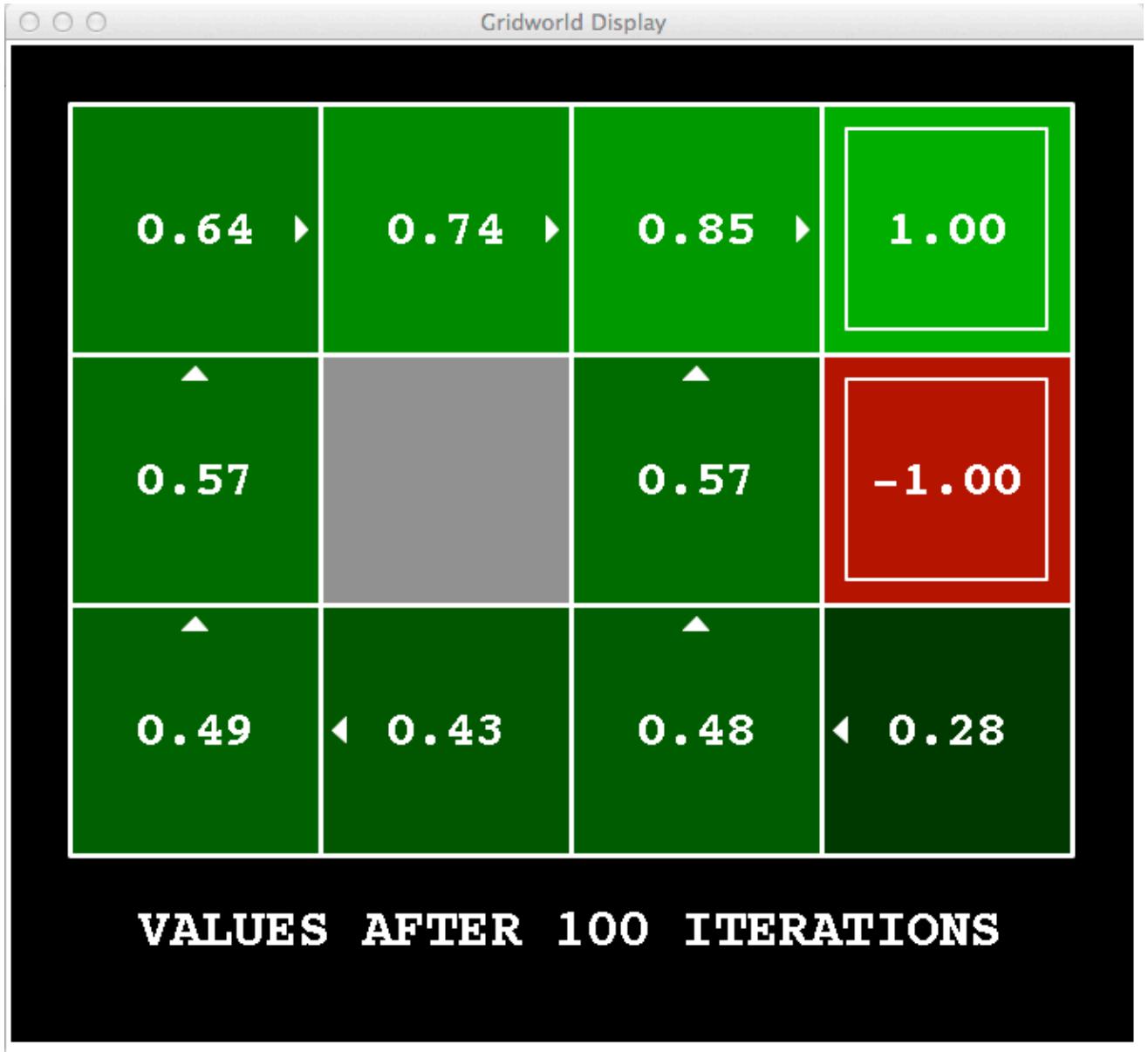
  $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally

- The optimal policy:

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

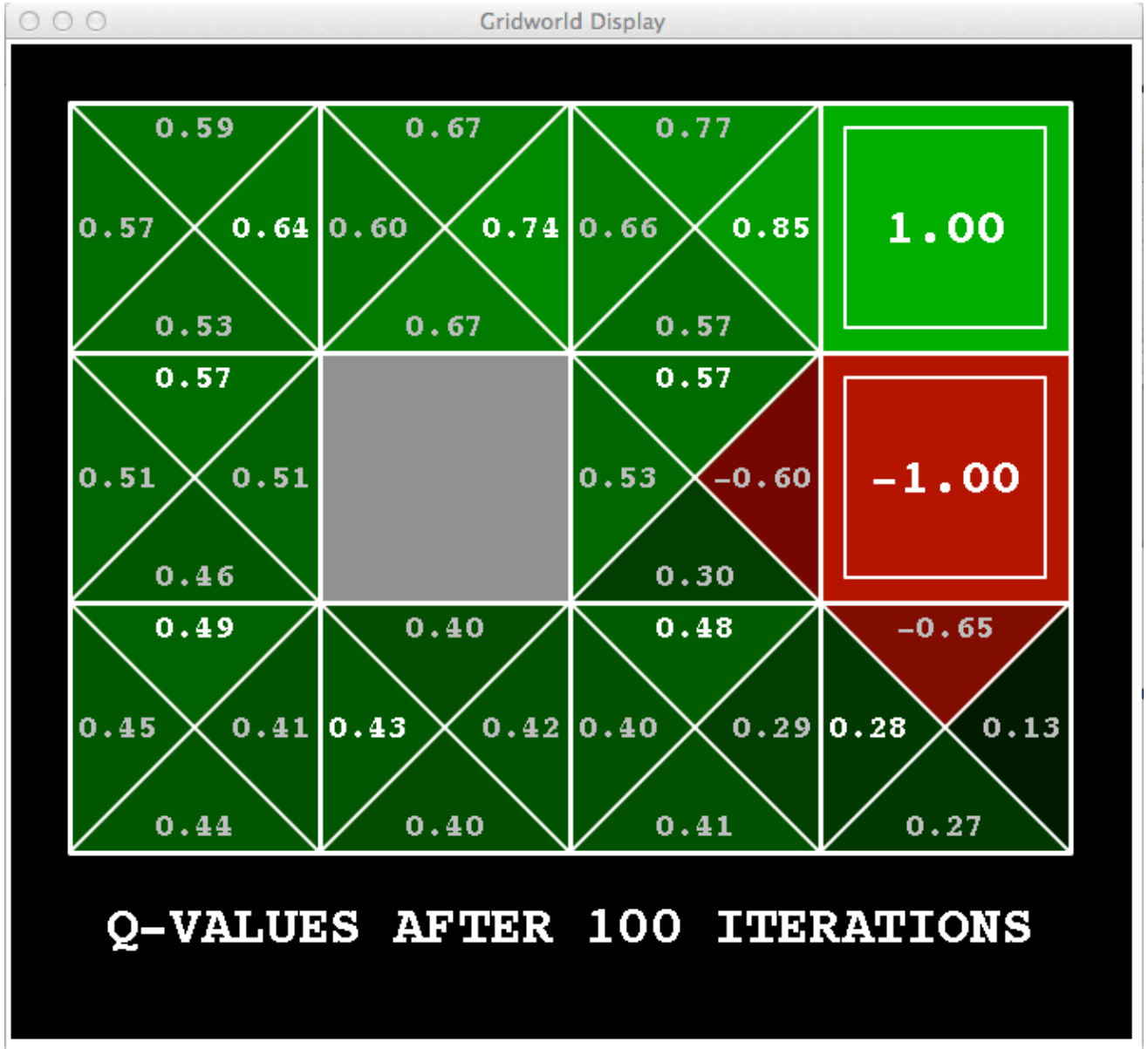


Snapshot of Demo – Gridworld V Values



Noise = 0.2
Discount Υ= 0.9
Living reward r = 0

Snapshot of Demo – Gridworld Q Values



Noise = 0.2
Discount = 0.9
Living reward = 0

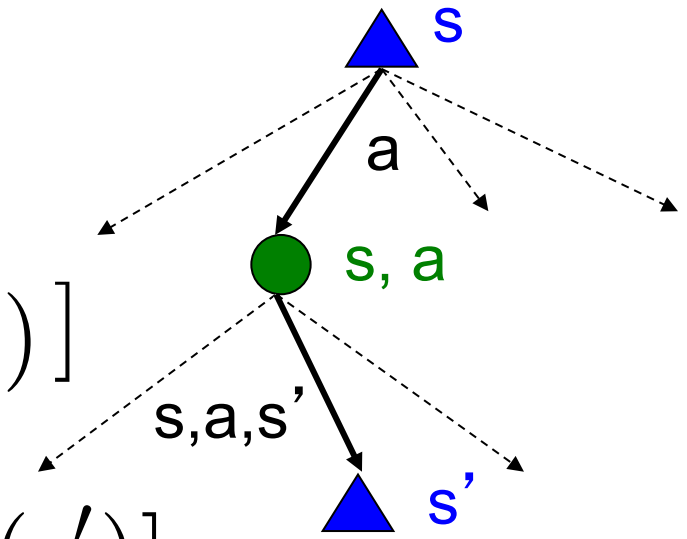
Values of States – Bellman Equation

- Recursive definition of value:

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

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

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



Immediate living award in S'

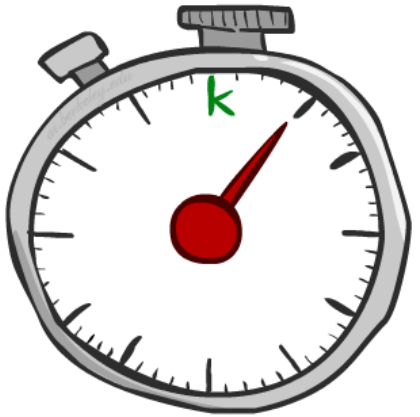
Recursive discounted future awards

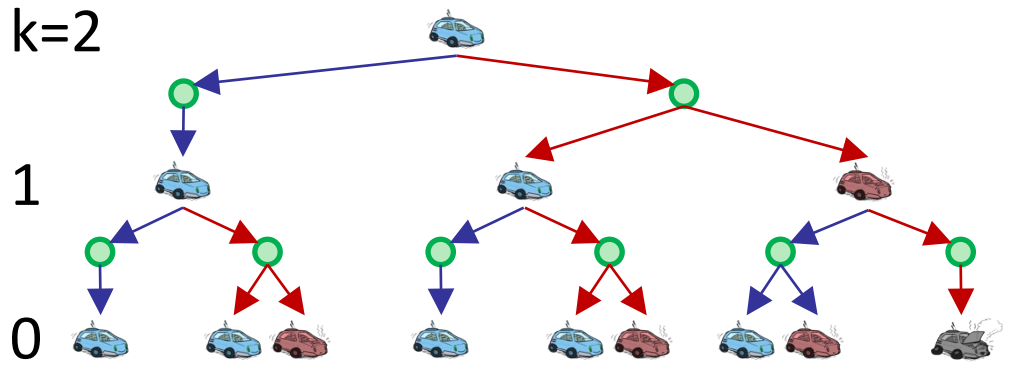
Expected Utility of an action a

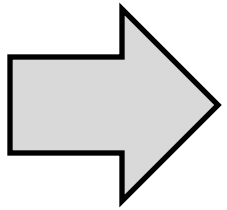
The action that has the max expected utility

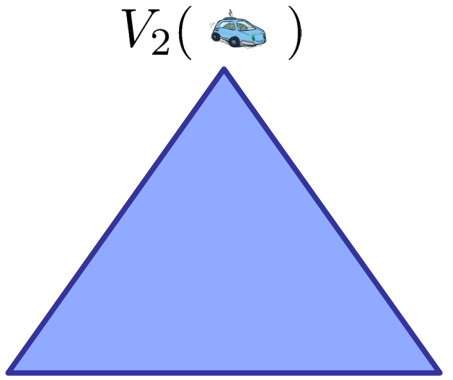
Time-Limited Values

- 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
  - k is the number of the time steps remaining in the future.



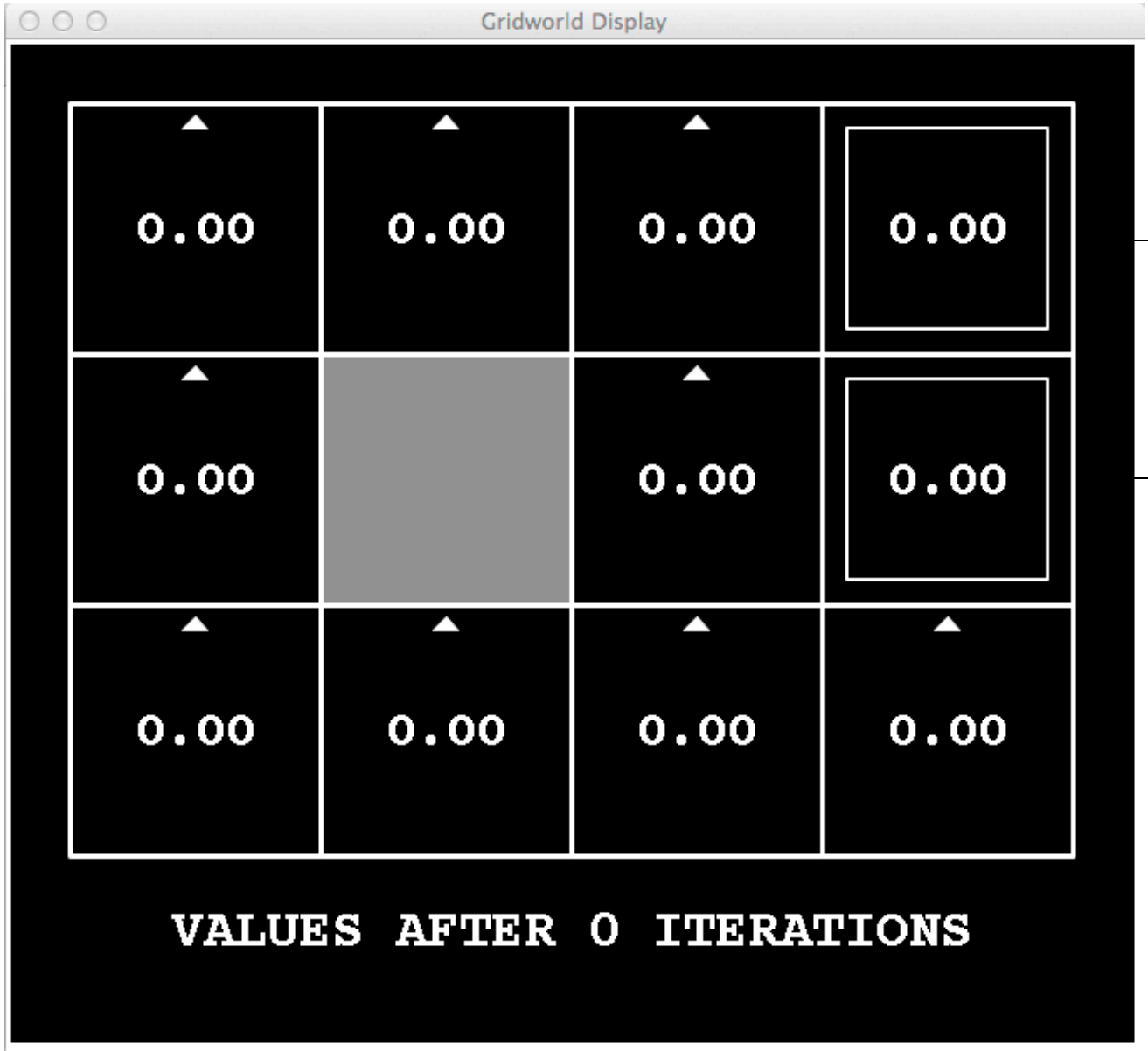






k=0

$\forall s, V(s)=0$



R(Exit1)

Exit1 → +1

R(Exit 2)

Exit2 → -1

Noise = 0.2
Discount = 0.9
Living reward = 0

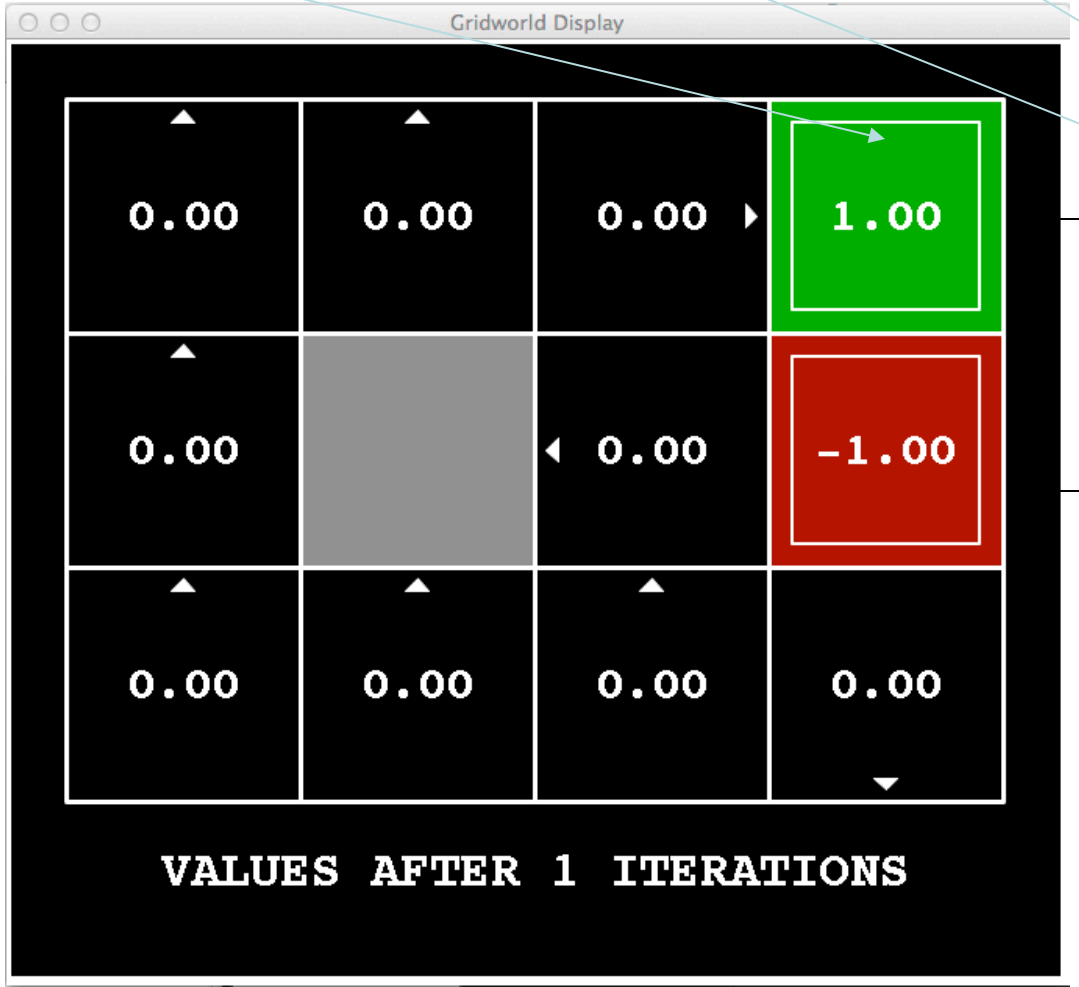
k=1

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

T R ϒ V

V*(4,3)=1.0 × [1 + 0.9 × 0] = 1.00

V*(4, 2)=1.0 × [-1 + 0.9 × 0] = -1.00



R(exit 1)

Exit (1.0)

+1

R(exit 2)

Exit (1.0)

-1

Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

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

T R ϒ V(4,3)

V*(3,3)=0.8 × [0 + 0.9 × (+1)] = 0.72

Noise = 0.2
Discount = 0.9
Living reward = 0

(3, 3) 0.72

V*(3,3) = 0.72, a = Right.

Up

Down

Right

Left

0.09

0.09

0.72

0

0.1 0.8 0.1

0.1 0.8 0.1

0.1 0.8 0.1

0.1 0.8 0.1

(2,3) 0

(3,3) 0

(4,3) 0

(2,3) 0

(3,2) 0

(4,3) 0

(3,3) 0

(4,3) 0

(3,2) 0

(3,3) 0

(2,3) 0

(3, 2) 0

0.1 × [0 + 0.9 × 0] = 0

0.8 × [0 + 0.9 × 0] = 0

0.1 × [0 + 0.9 × 1] = 0.09

0+0+0.09= 0.09

0.1 × [0 + 0.9 × 0] = 0

0.8 × [0 + 0.9 × 1] = 0.72

0.1 × [0 + 0.9 × 0] = 0

0+0.72+0= 0.72

0.1 × [0 + 0.9 × 0] = 0

0.8 × [0 + 0.9 × 0] = 0

0.1 × [0 + 0.9 × 0] = 0

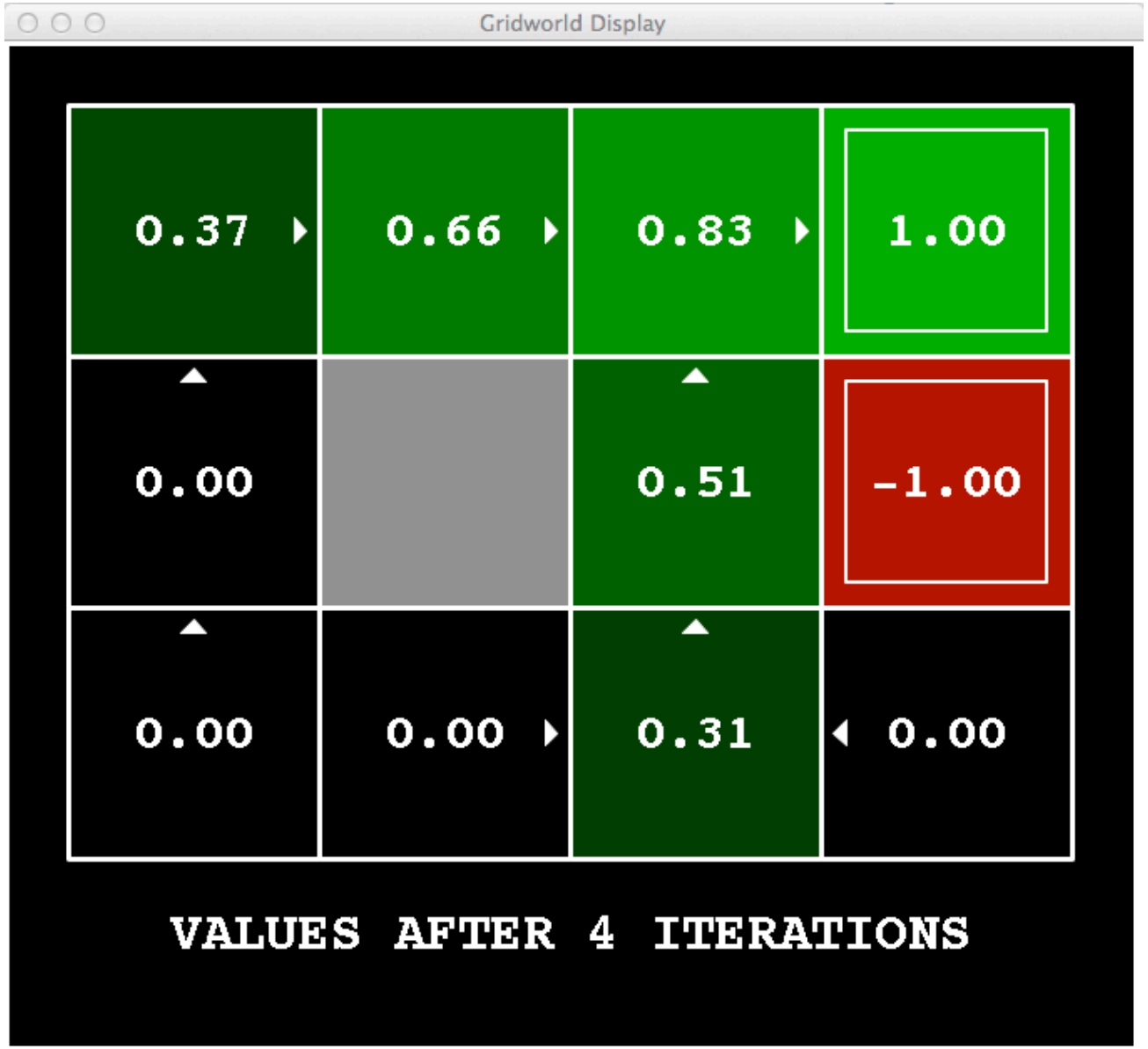
0+0+0= 0

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

k=3

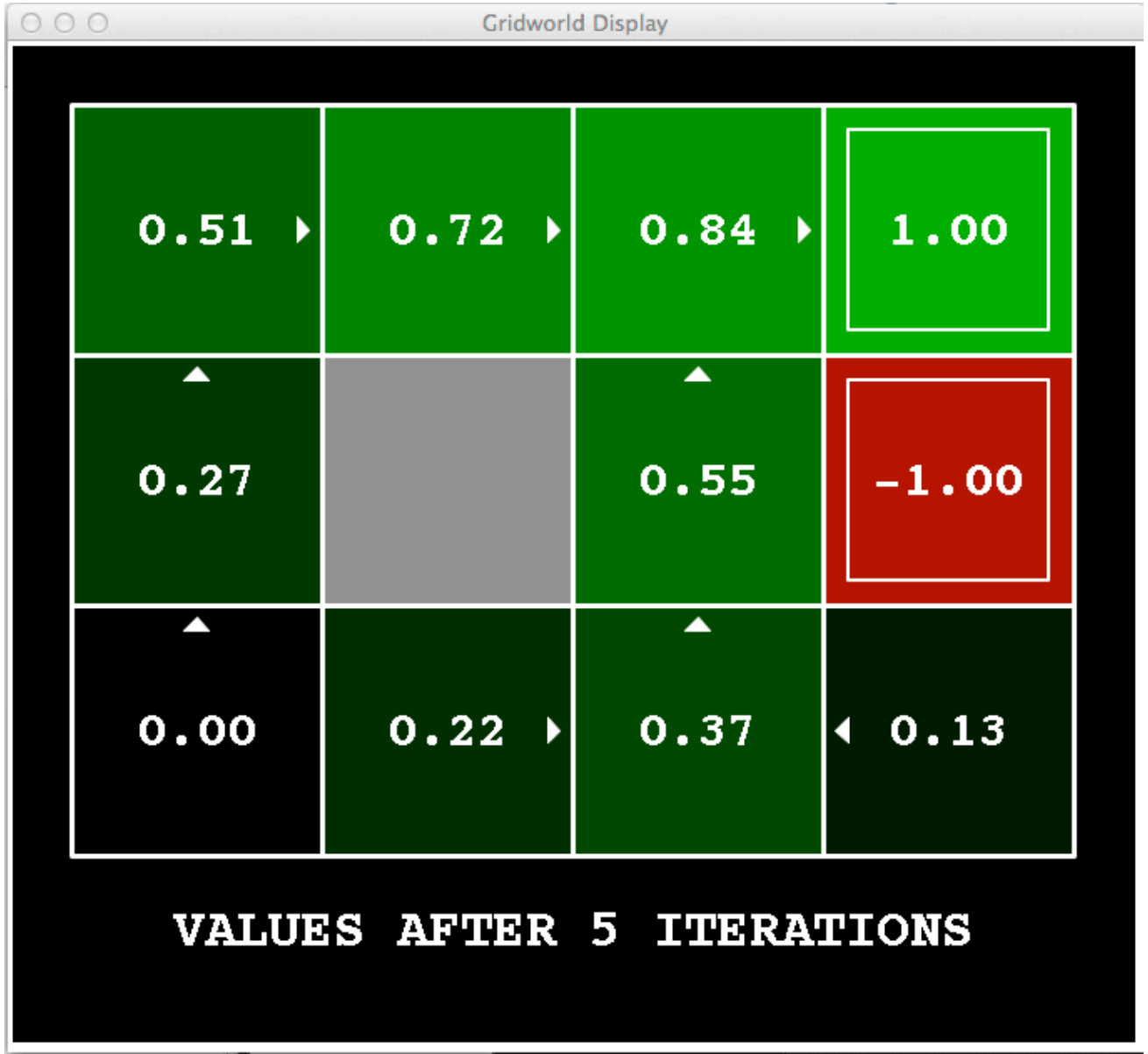
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



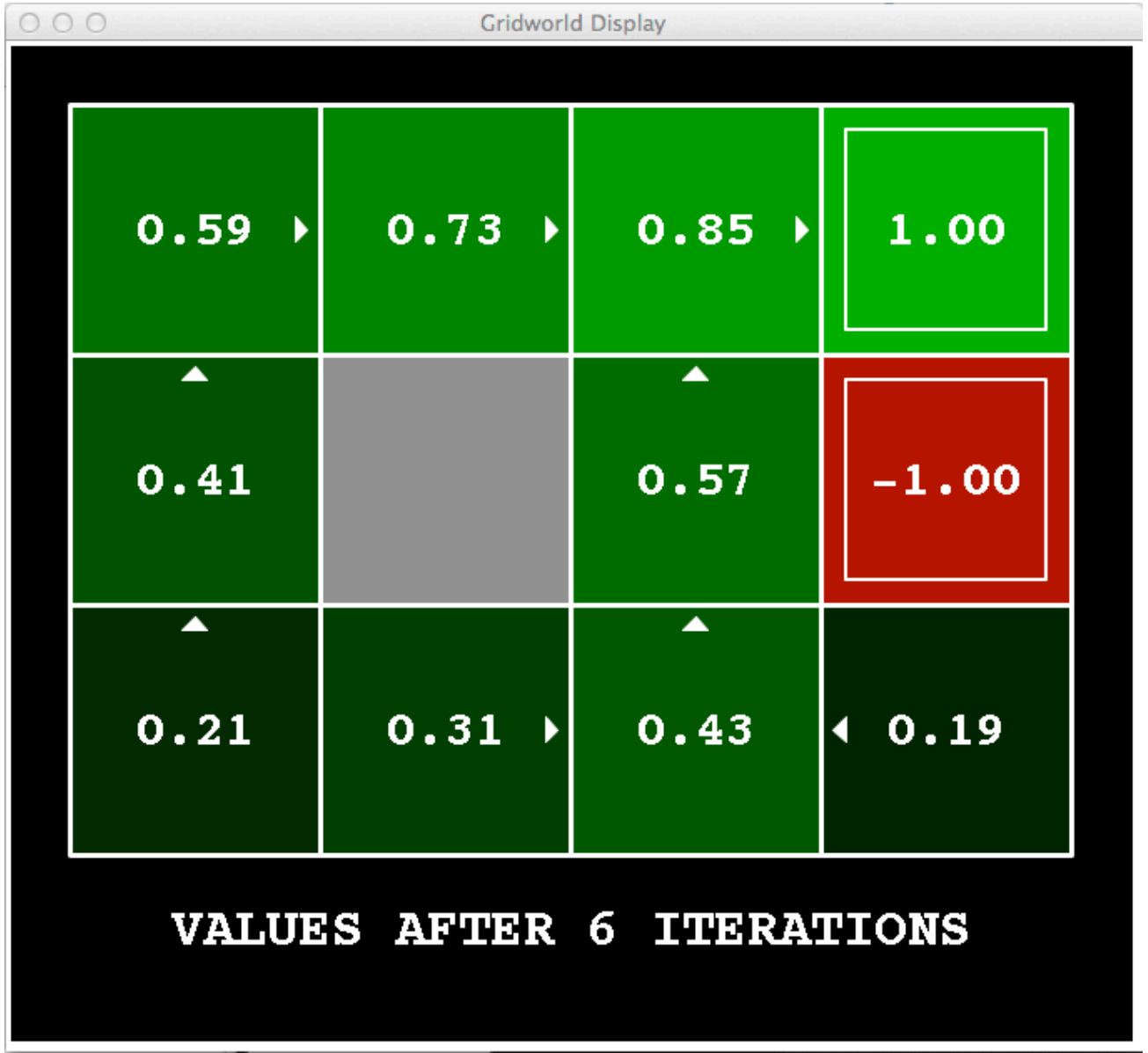
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



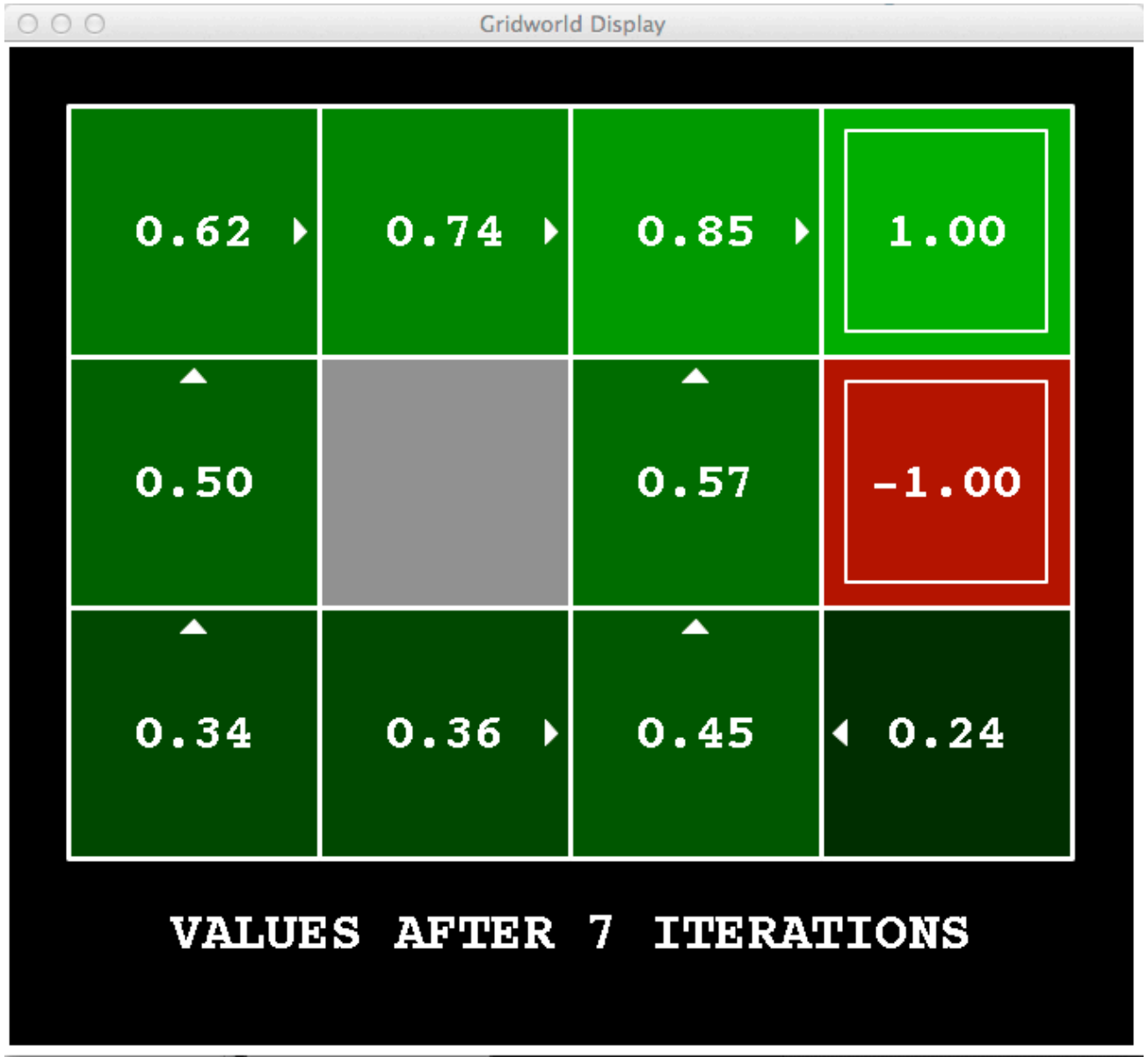
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



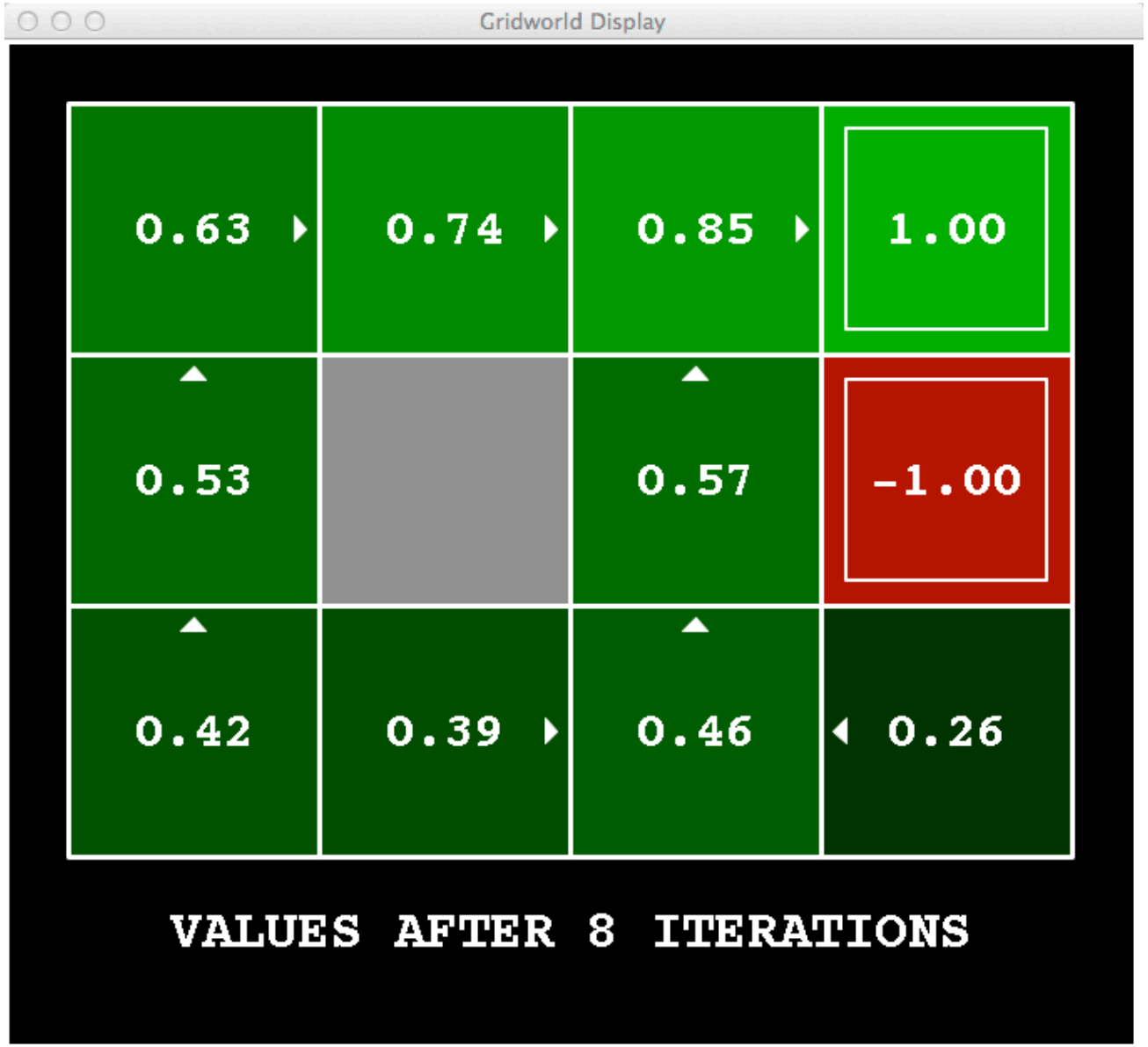
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



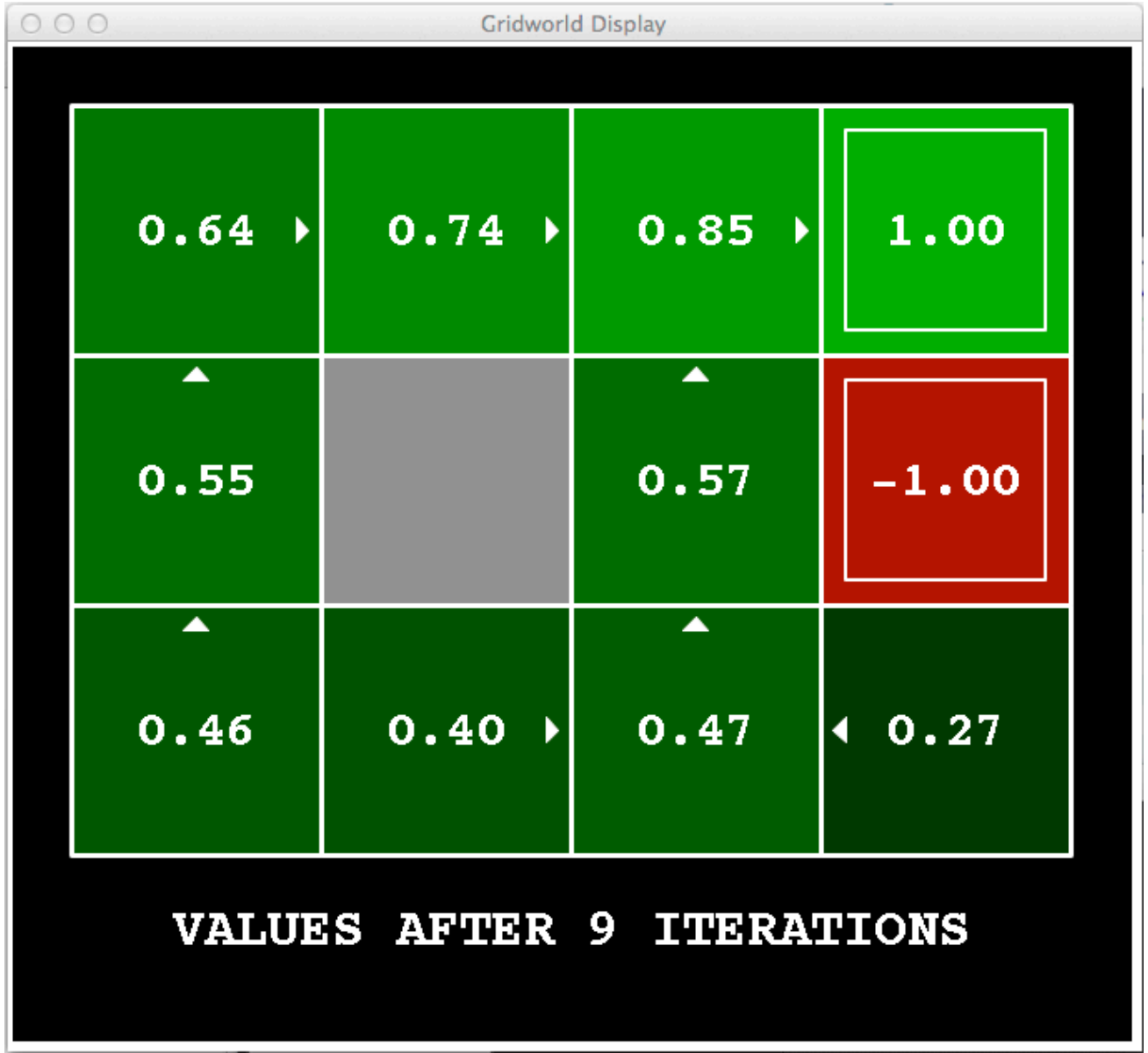
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



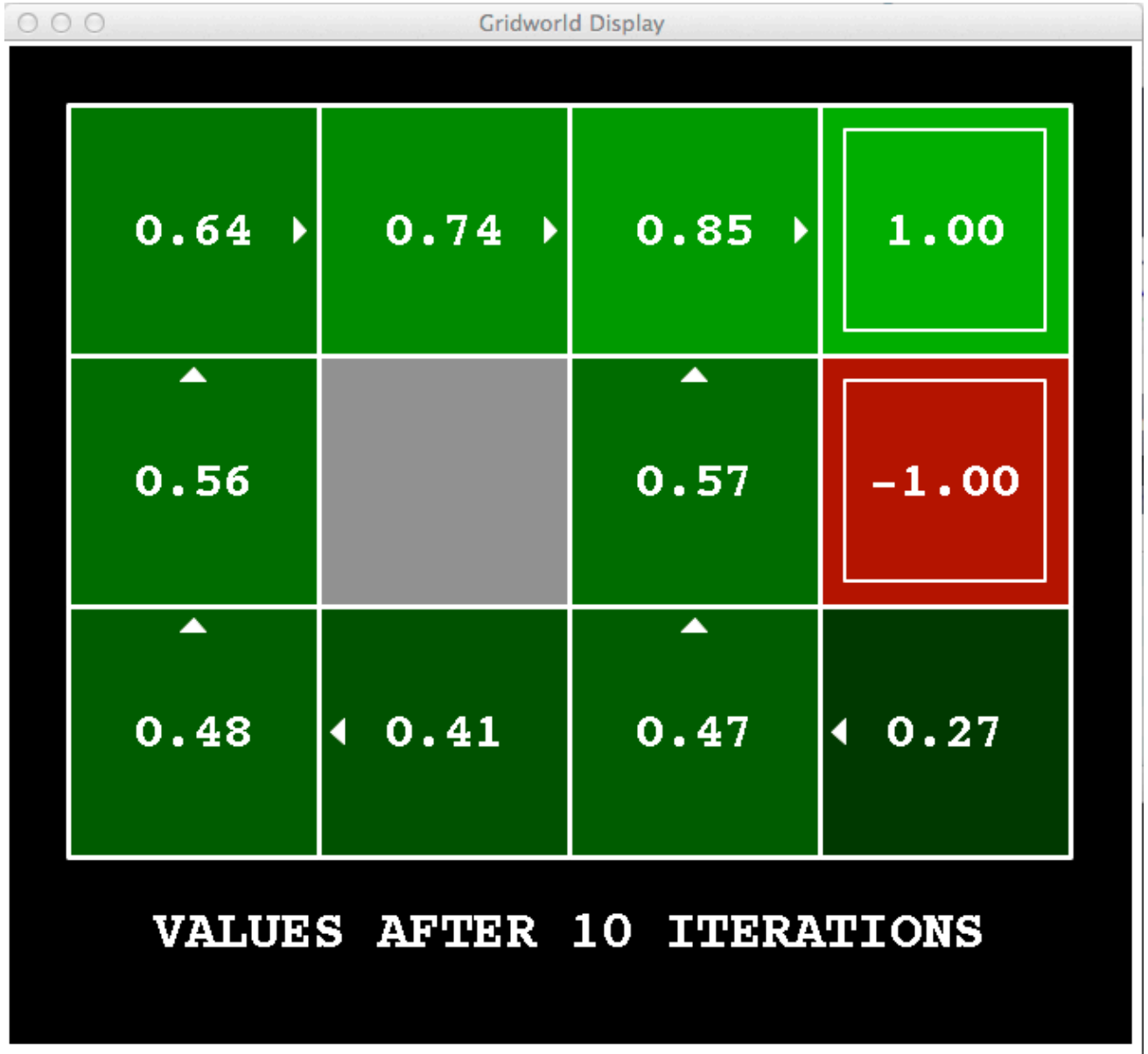
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



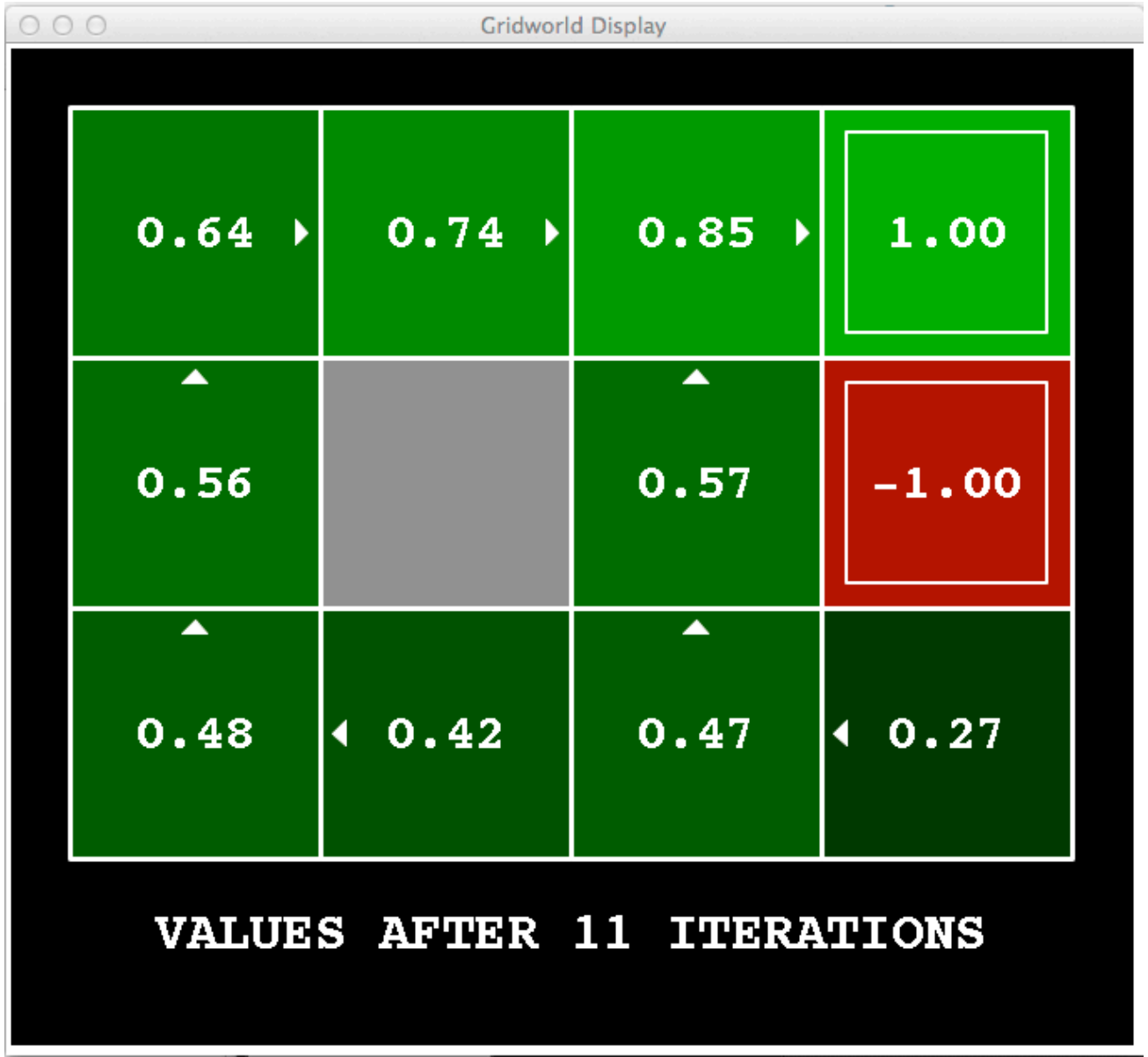
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



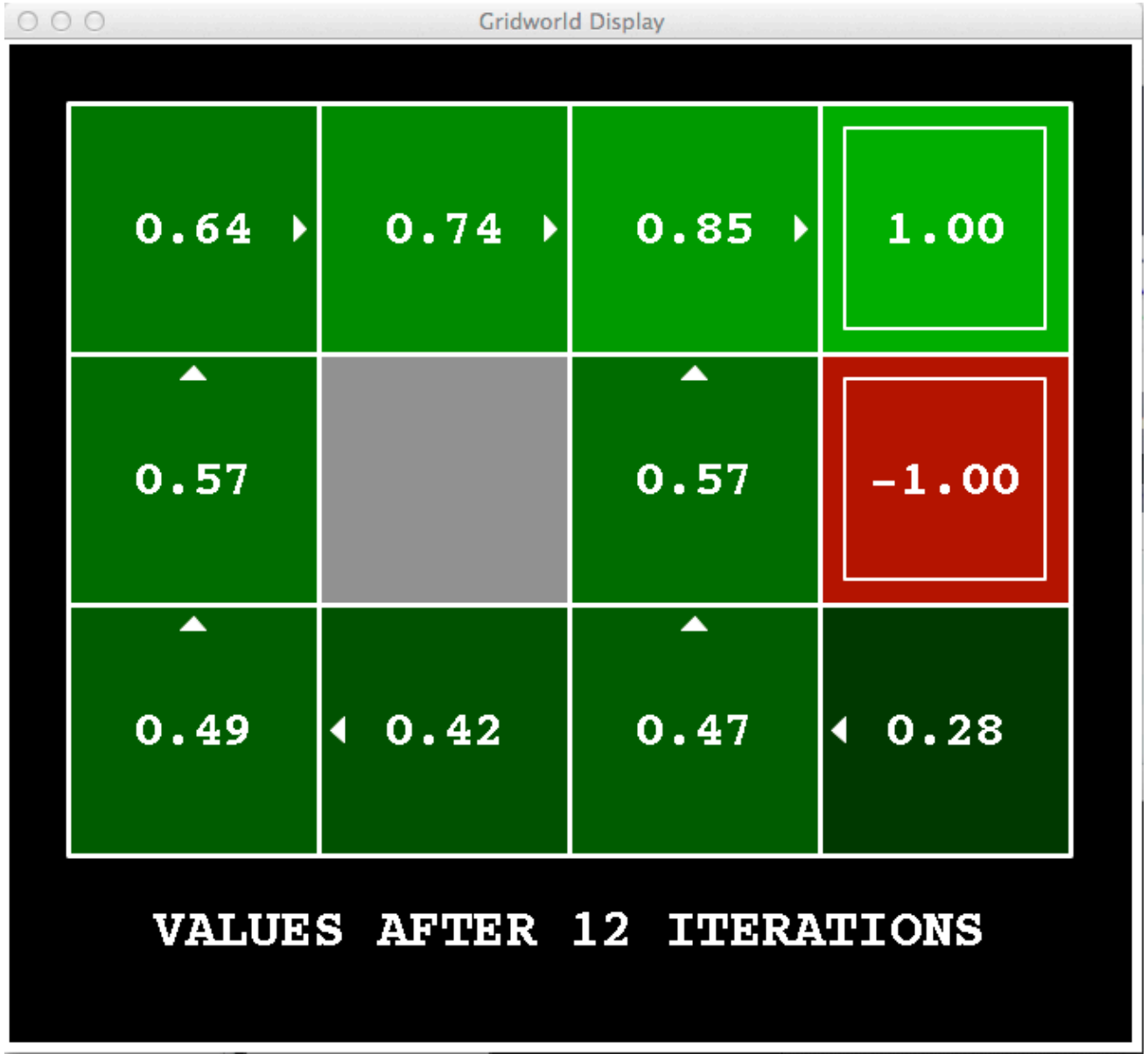
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



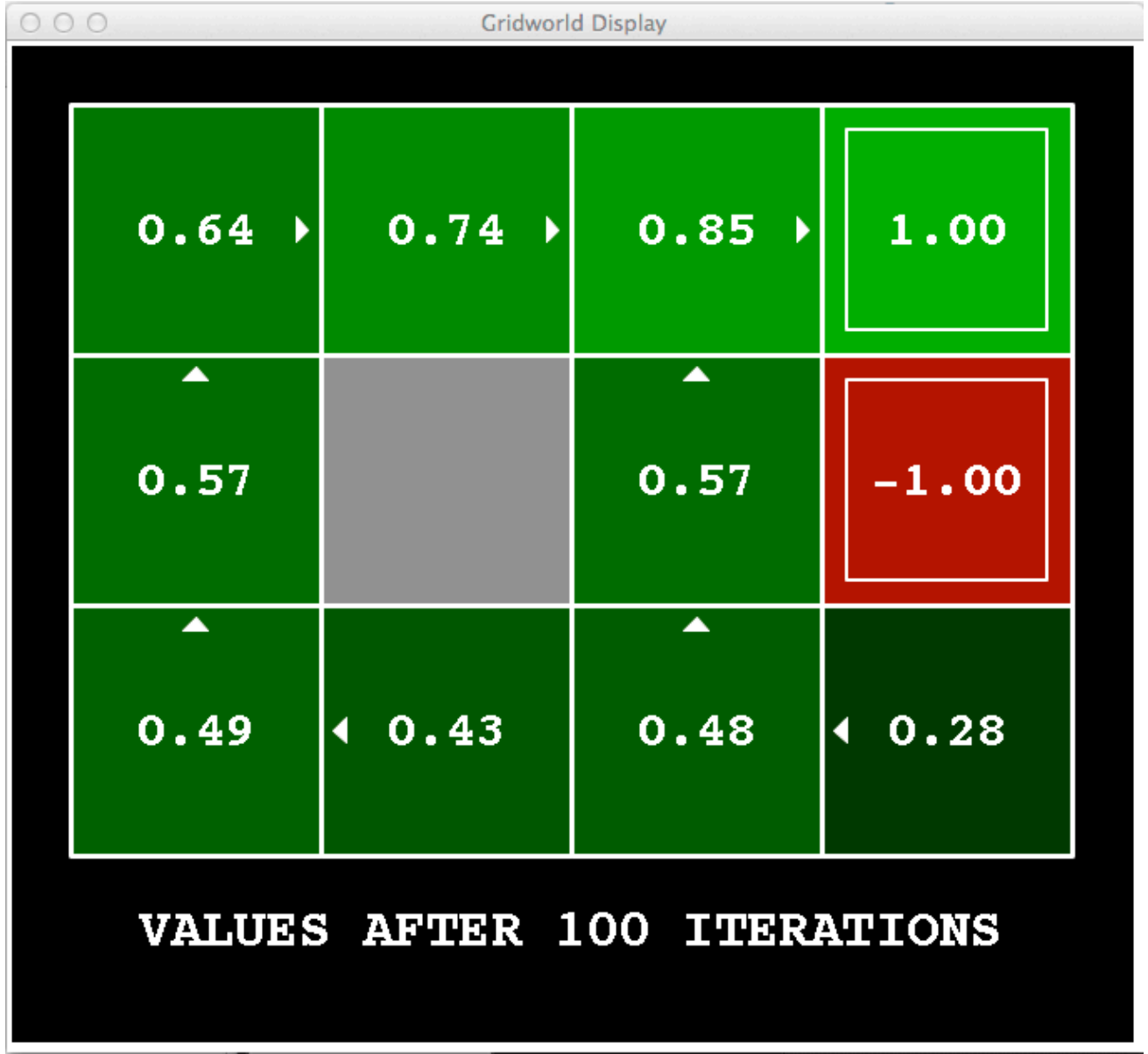
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



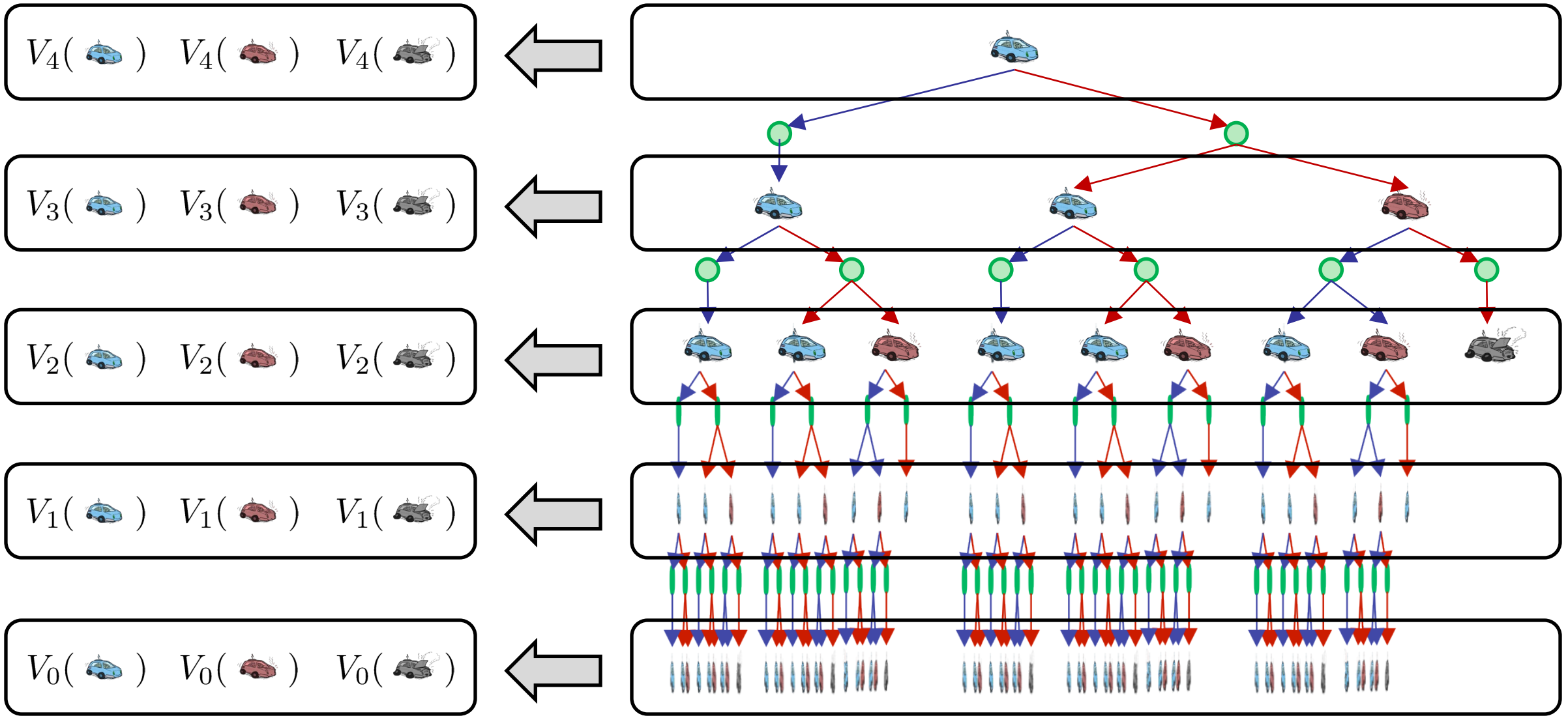
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



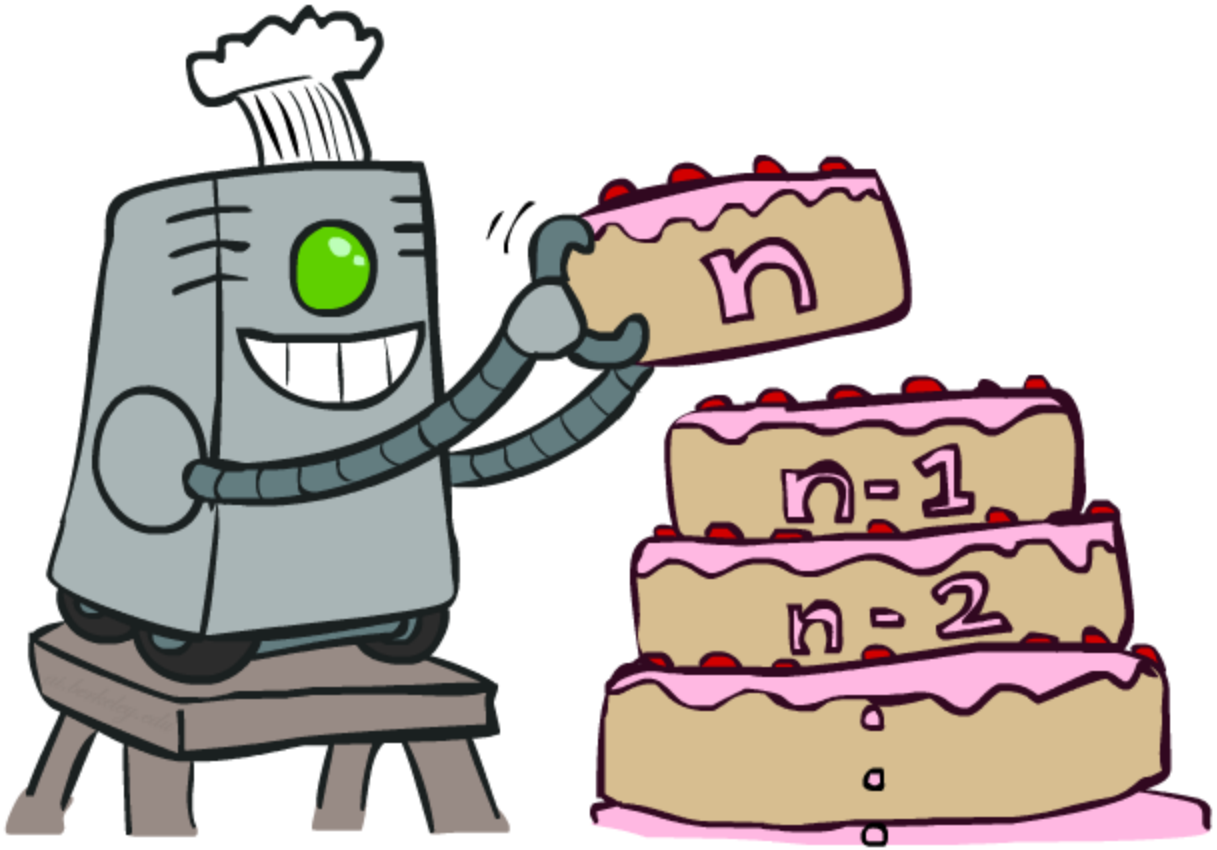
Noise = 0.2
Discount = 0.9
Living reward = 0

Computing Time-Limited Values



Value Iteration

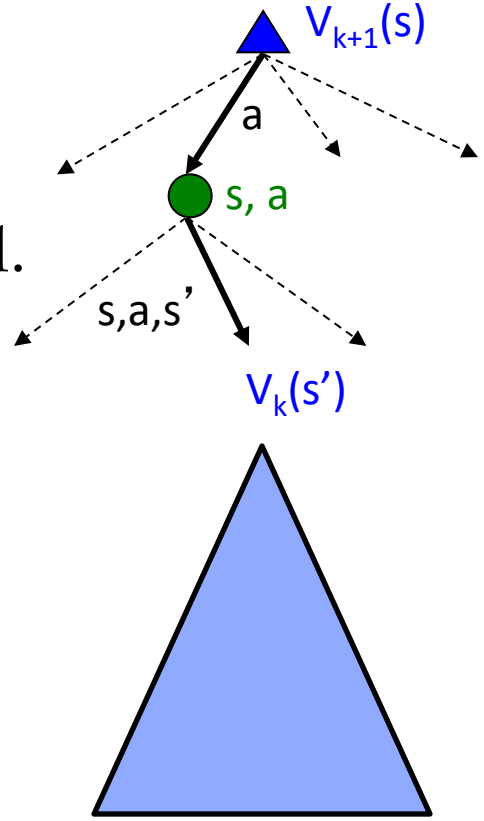
Idea: Find the optimal policy by iteratively updating the value function by the Bellman equation.



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 **Value Iteration** from each state:

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



- Stopping condition: when the max difference between two successive value function estimates is below a small threshold.
- Complexity of each iteration: O(s^2a)
  - each state is O(sa)
  - iterate for s states
  - so the total is 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

Example: Value Iteration

V_2

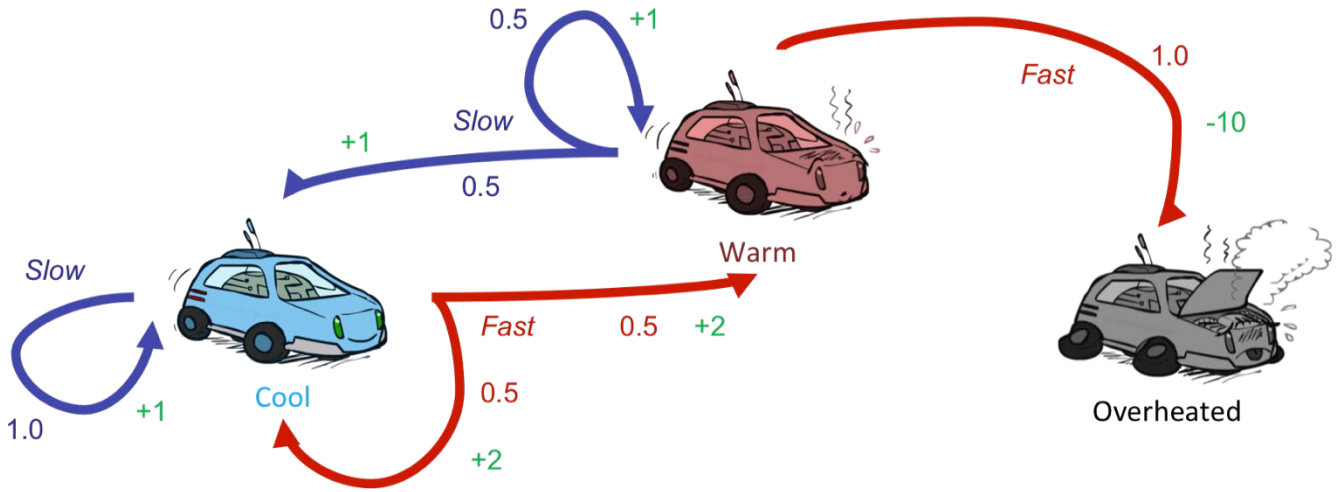
V_1

S: 1

F: .5*2+.5*2=2

V_0

0 0 0



γ=1 Assume no discount!

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

Example: Value Iteration

V_2

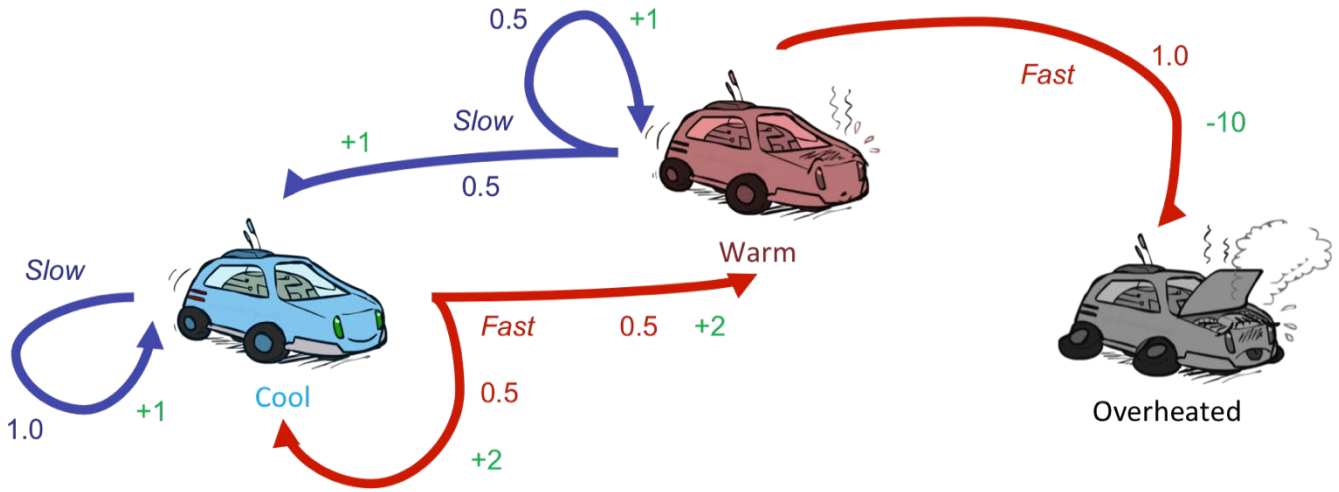
V_1

2
Fast

S: .5*1+.5*1=1
F: -10

V_0

0 0 0



γ=1 Assume no discount!

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

Example: Value Iteration

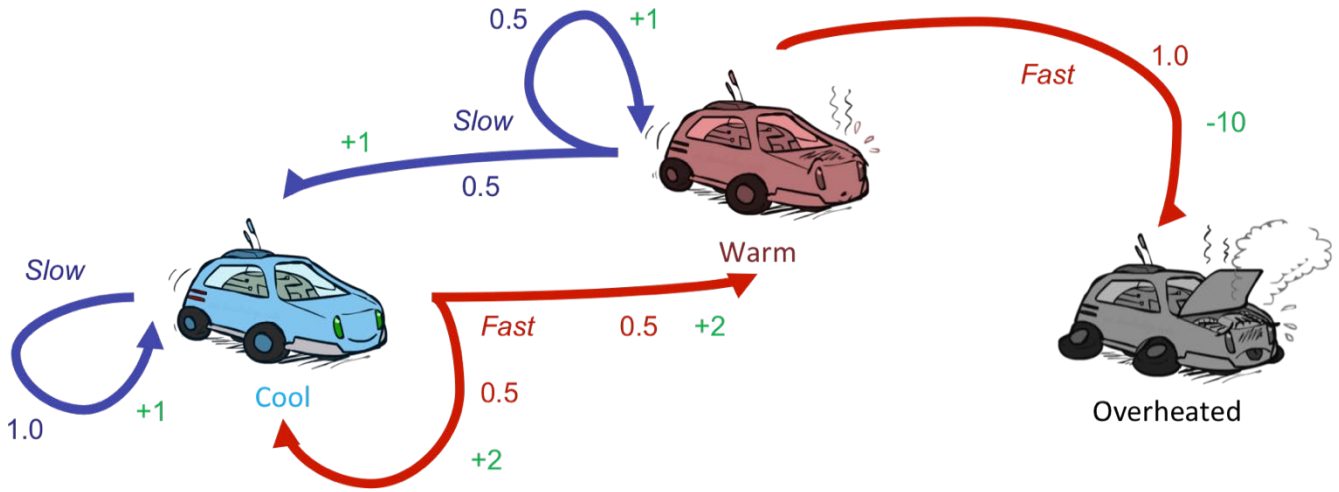
V_2

V_1

2	1	0
Fast	Slow	

V_0

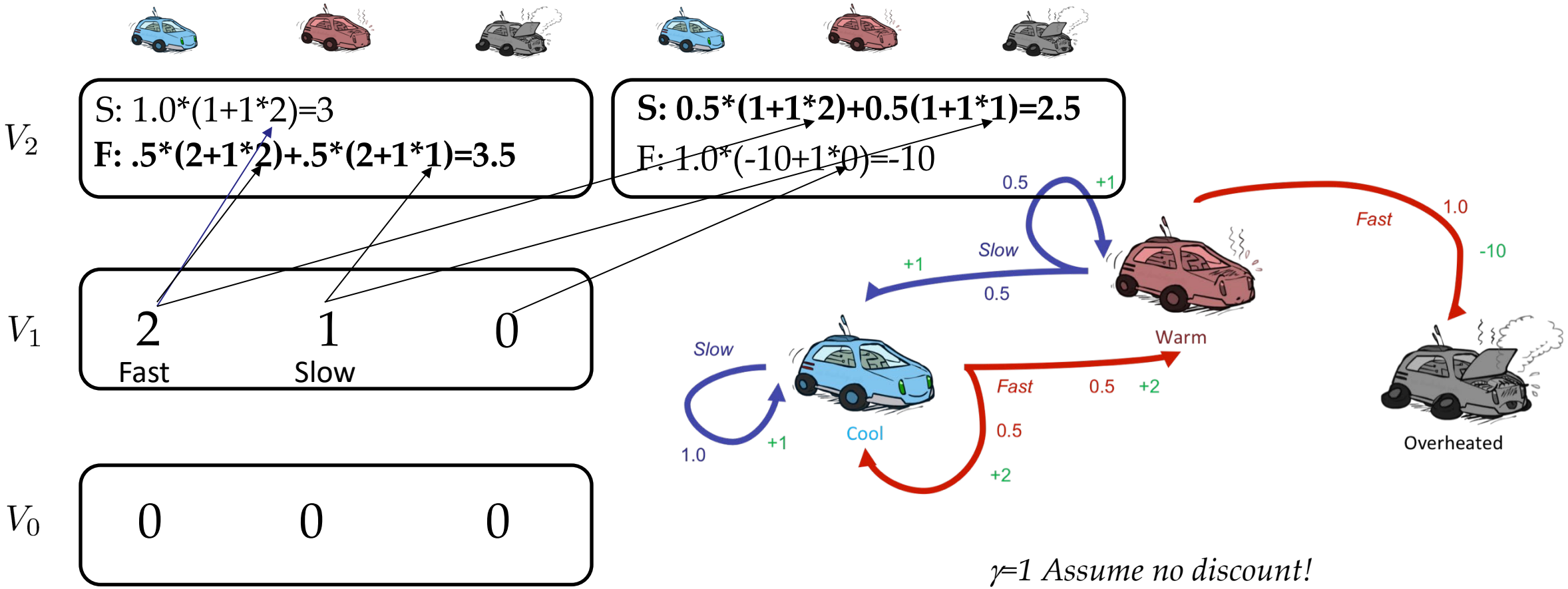
0	0	0



γ=1 Assume no discount!

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

Example: Value Iteration

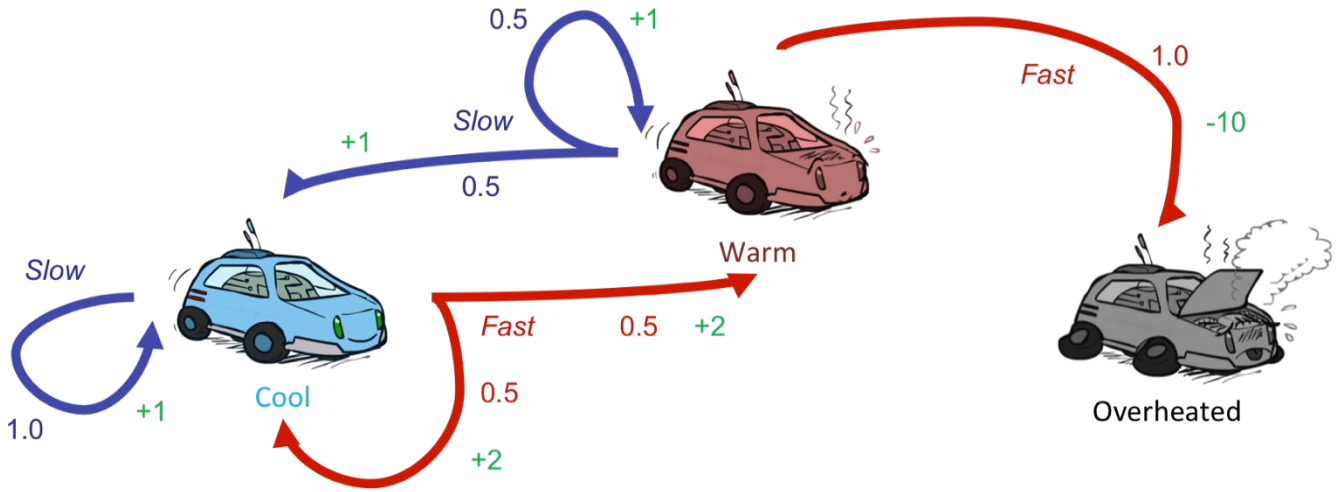


γ=1 Assume no discount!

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

Example: Value Iteration

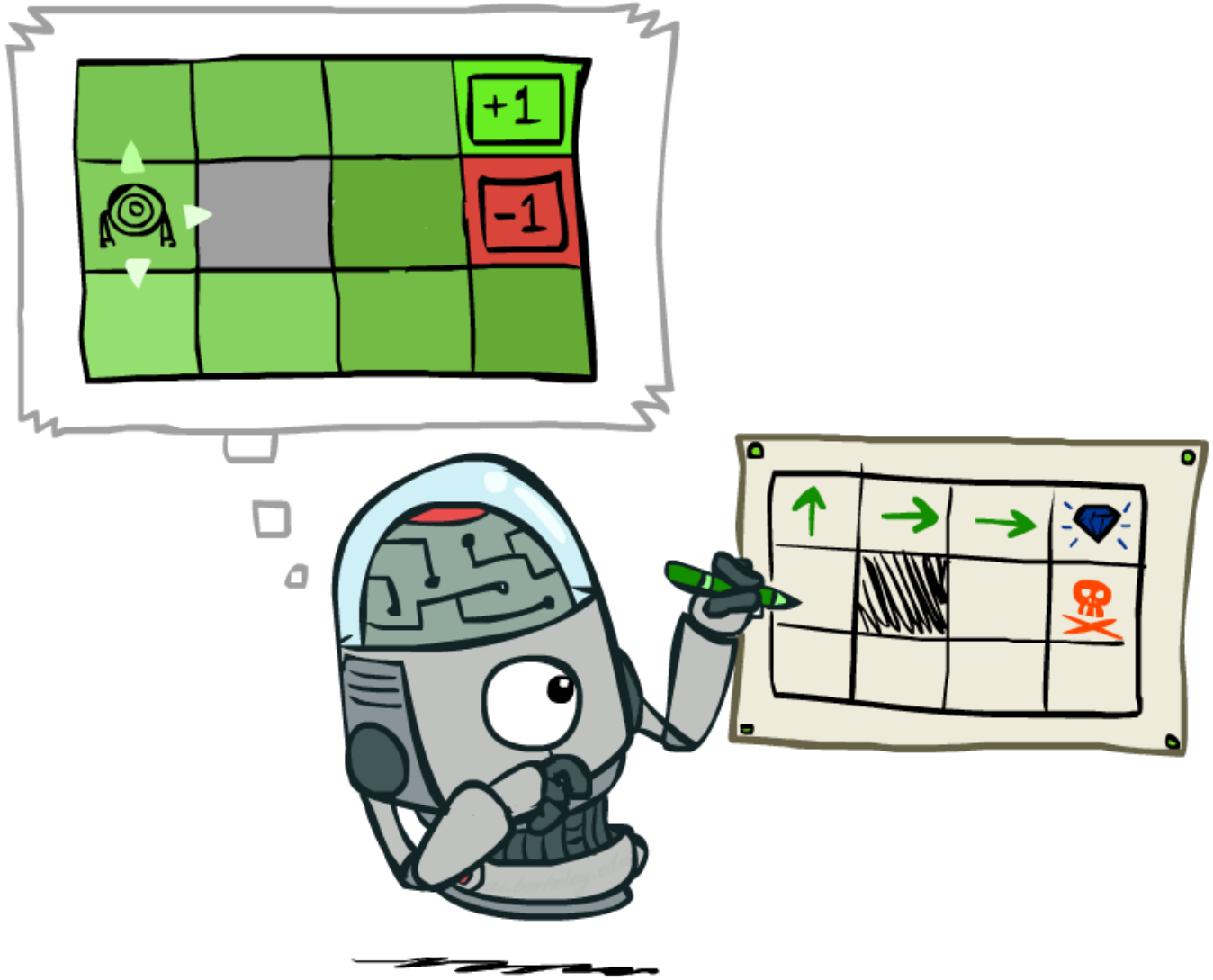
V_2	3.5 Fast	2.5 Slow	0
V_1	2 Fast	1 Slow	0
V_0	0	0	0



γ=1 Assume no discount!

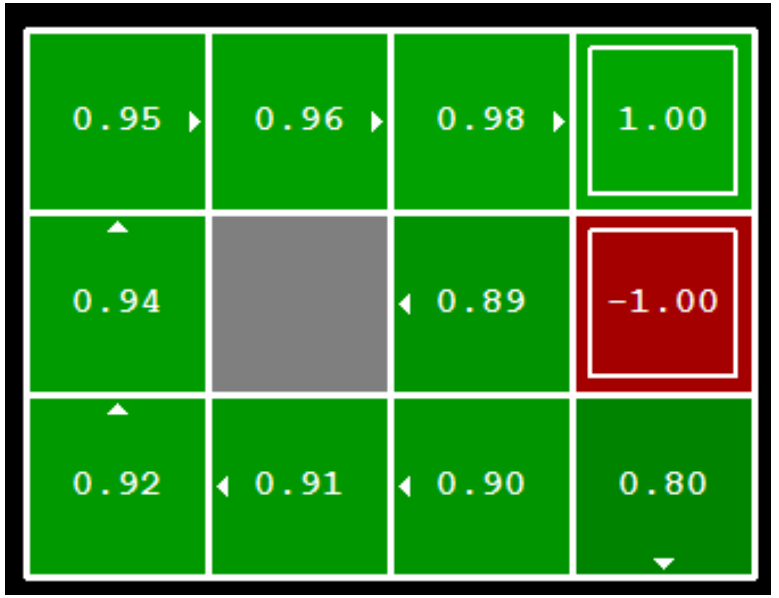
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Policy Extraction



Computing Actions from Values

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

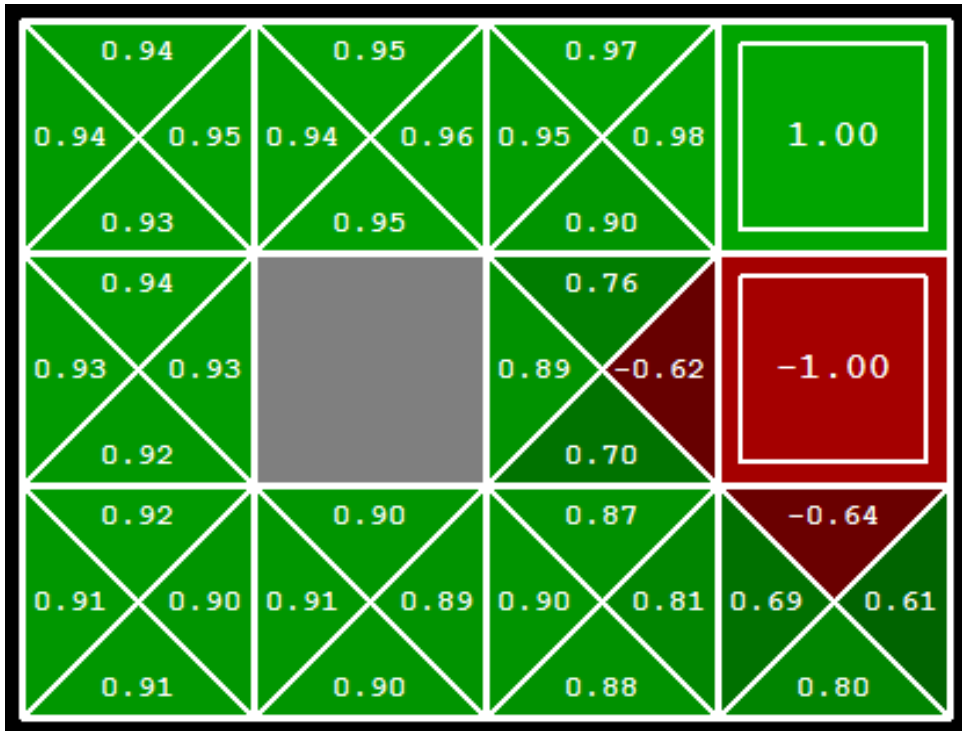


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

Computing Actions from Q-Values

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



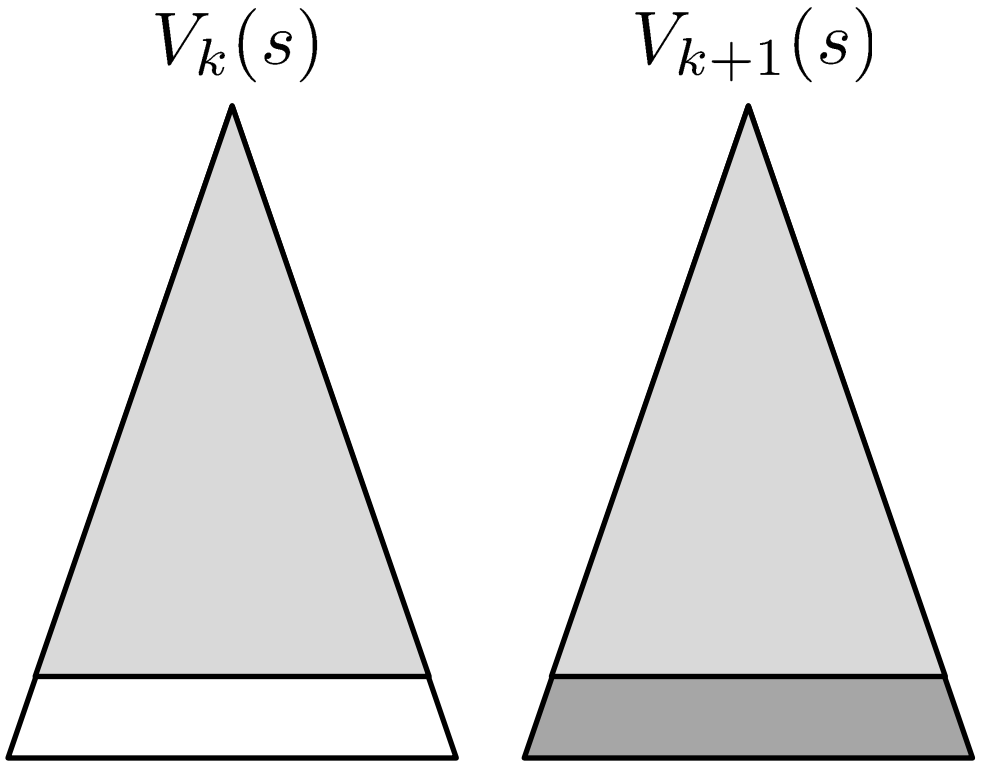
- How should we act?
  - Completely trivial to decide!

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

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

Convergence

- How do we know the V_k vectors are going to converge?
- Case 1: If γ=1 (no discount), the tree has maximum depth M, then V_M holds the actual untruncated values.
- Case 2: If γ<1 (discounting)
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 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} , 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

Input: MDP M={S, A, T(s, a, s'), R(s, a, s')}
Output: Optimal policy π^*

Set V to arbitrary value function, e.g., V(s)=0 for all s.
Repeat
    $\Delta = 0$ `//value difference between` V_{k+1} `and` V_k
    for each $s \in S$

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

    $\Delta = \max(\Delta, |\Delta - V^*(s)|)$
Until $\Delta < \theta$ (a small positive number)

Output an optimal policy π^* such that

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

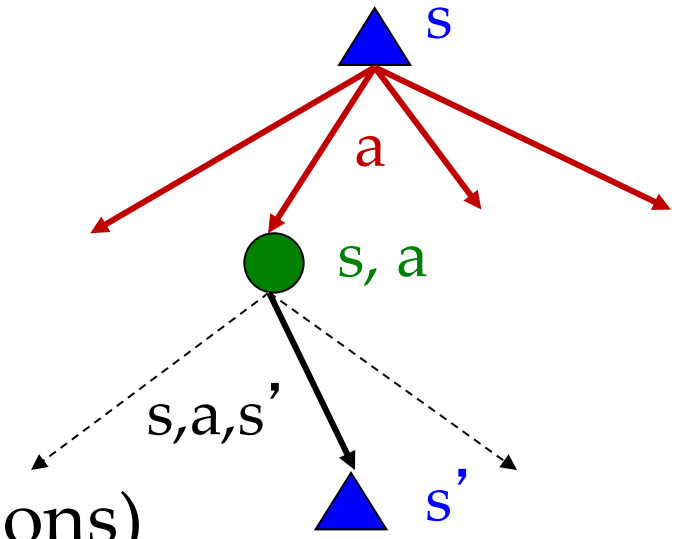
Let's Think

- Take a minute, think about value iteration.
- What is the biggest question you have about it?

Problems with Value Iteration

- Value iteration repeats the Bellman updates:

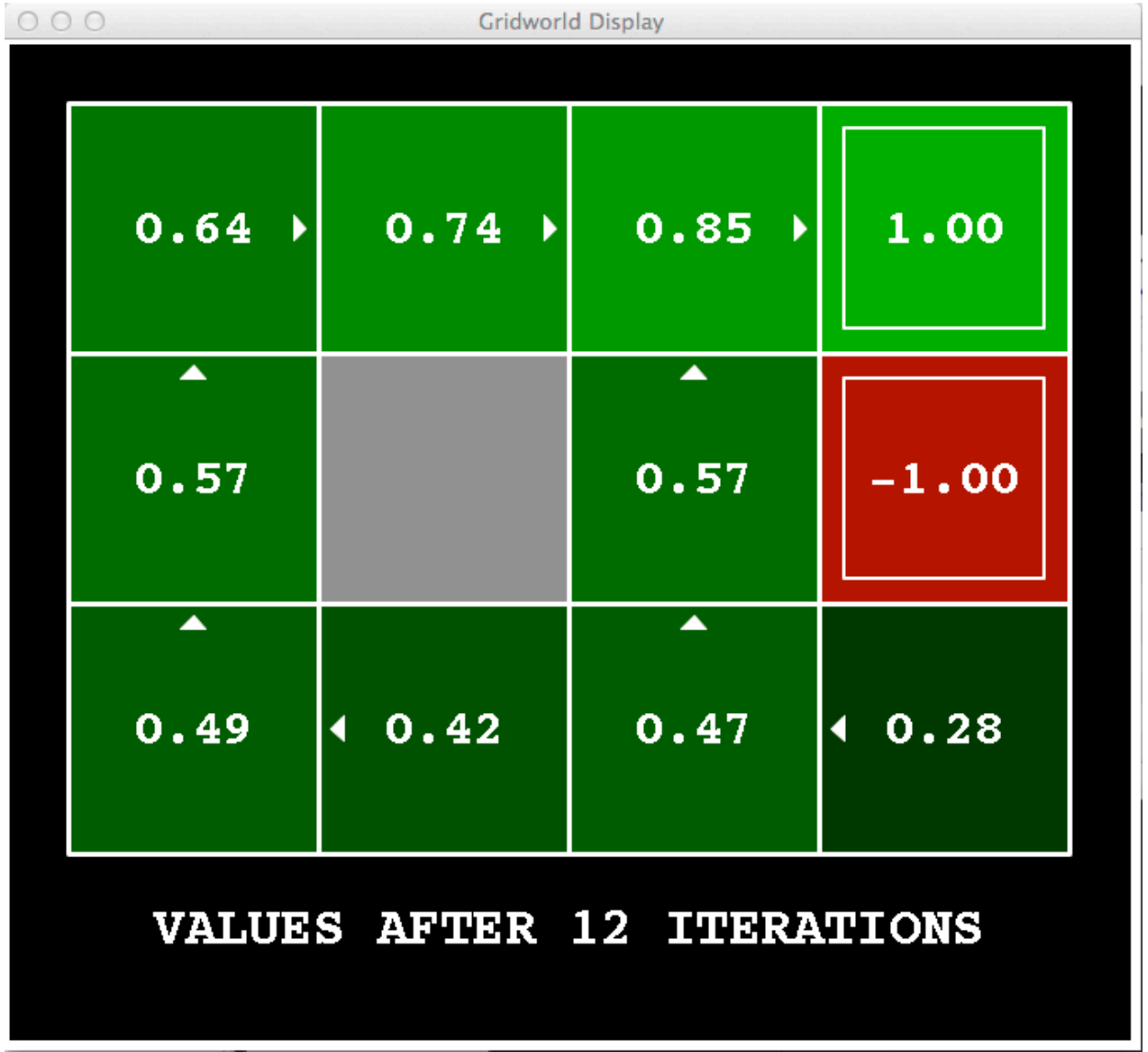
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

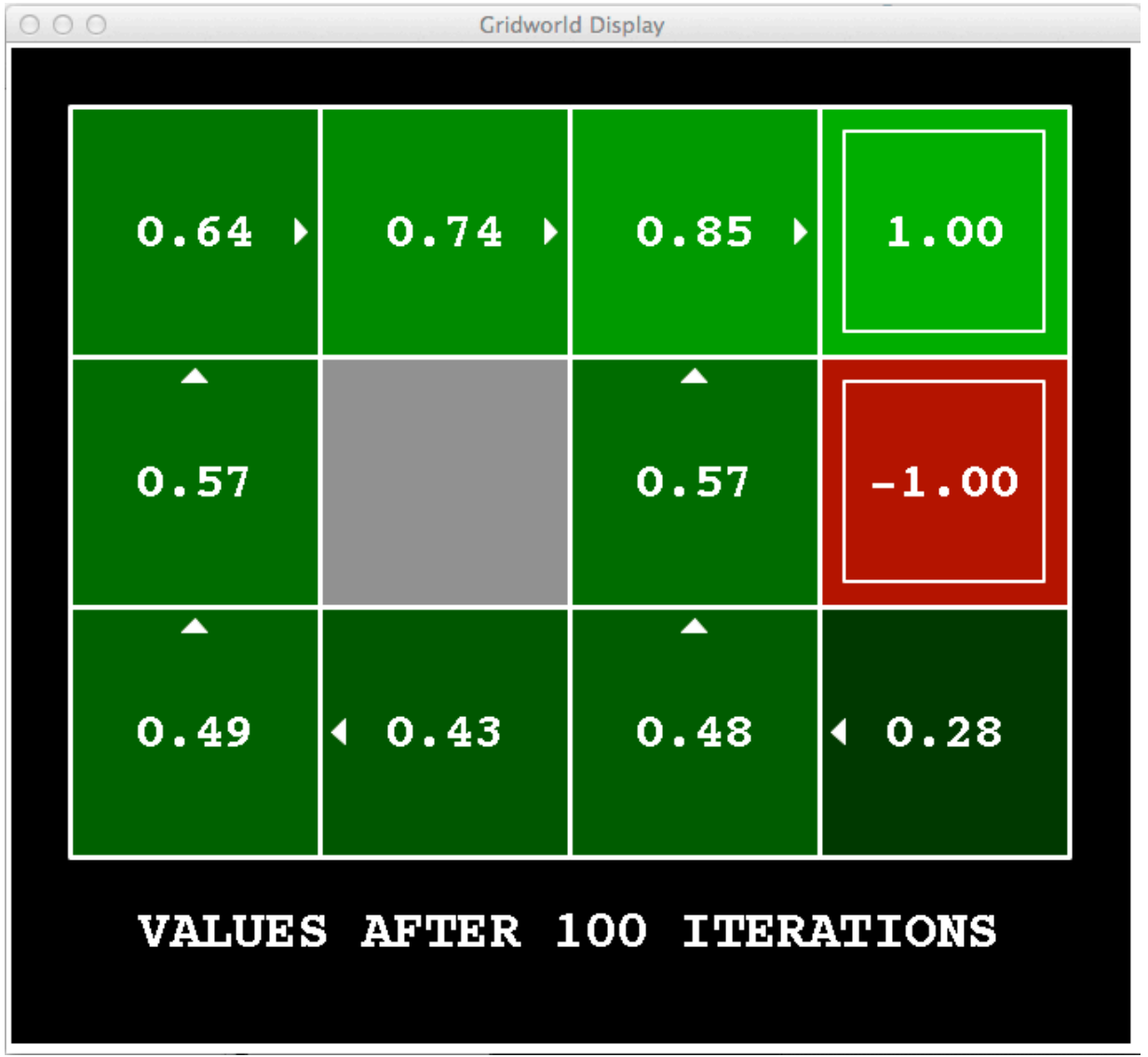


- Problem 1: It's slow – O(s^2a) per iteration (k iterations)
- Problem 2: The "max action" at each state rarely changes
- Problem 3: The policy often converges long before the values

k=12

k=100


Gridworld Display
0.64
0.74
0.85
1.00
0.57
0.57
-1.00
0.49
0.42
0.47
0.28
VALUES AFTER 12 ITERATIONS


Gridworld Display
0.64
0.74
0.85
1.00
0.57
0.57
-1.00
0.49
0.43
0.48
0.28
VALUES AFTER 100 ITERATIONS

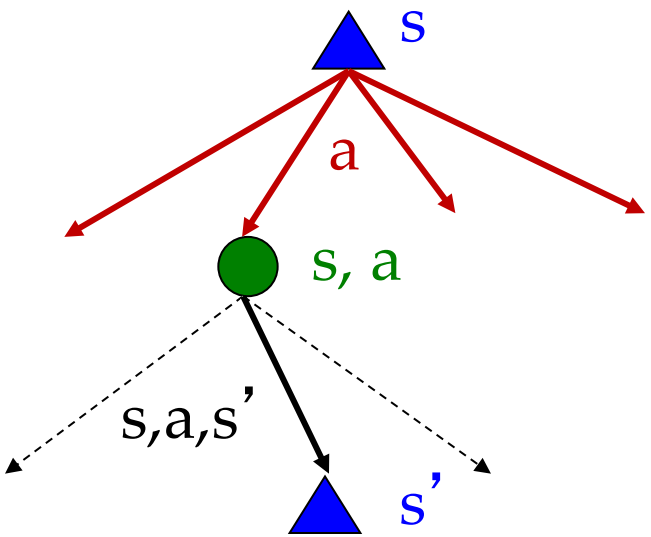
Policy Iteration

- 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.

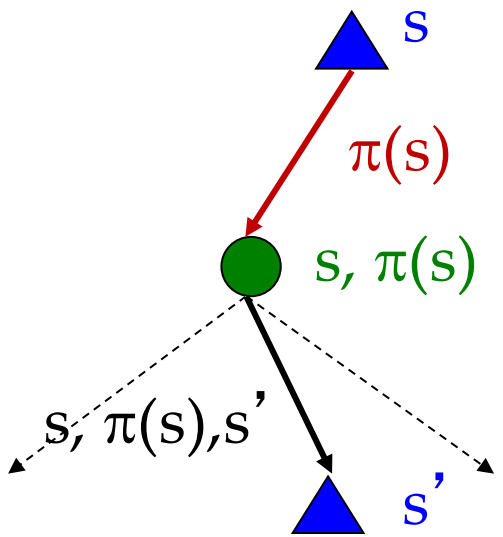
- This is policy iteration.
  - It's still optimal!
  - Can converge (much) faster under some conditions.

Fixed Policies

Do the optimal action



Do what π says to do



- Search 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

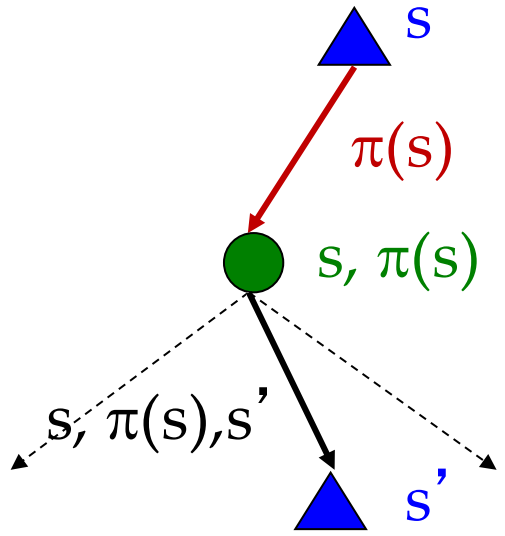
- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy.

- 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 (Bellman equation with a fixed policy π):

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



Policy Evaluation

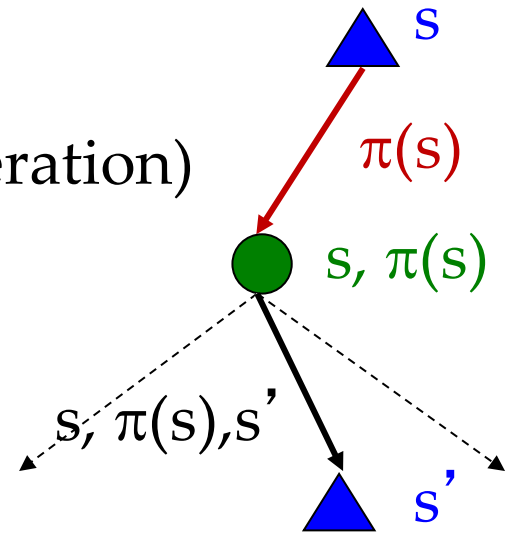
- How do we calculate the Vs for a fixed policy π?

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

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

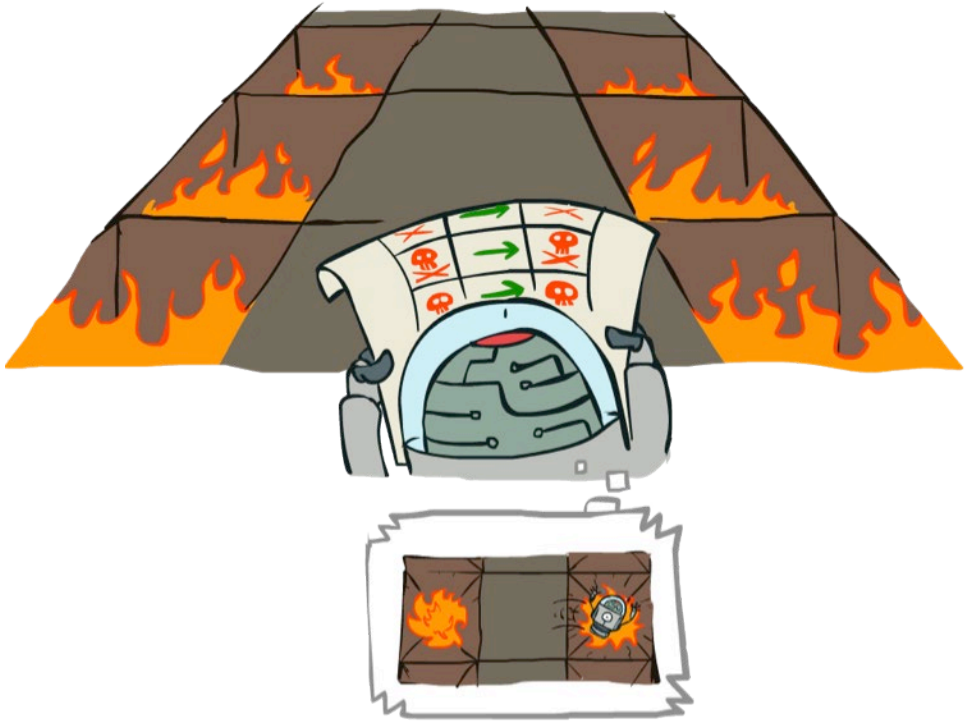
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

  - Without the maxes, the Bellman equations are just a linear system.
  - Efficiency: $O(S^2)$ per iteration because a=1 (one action per state).



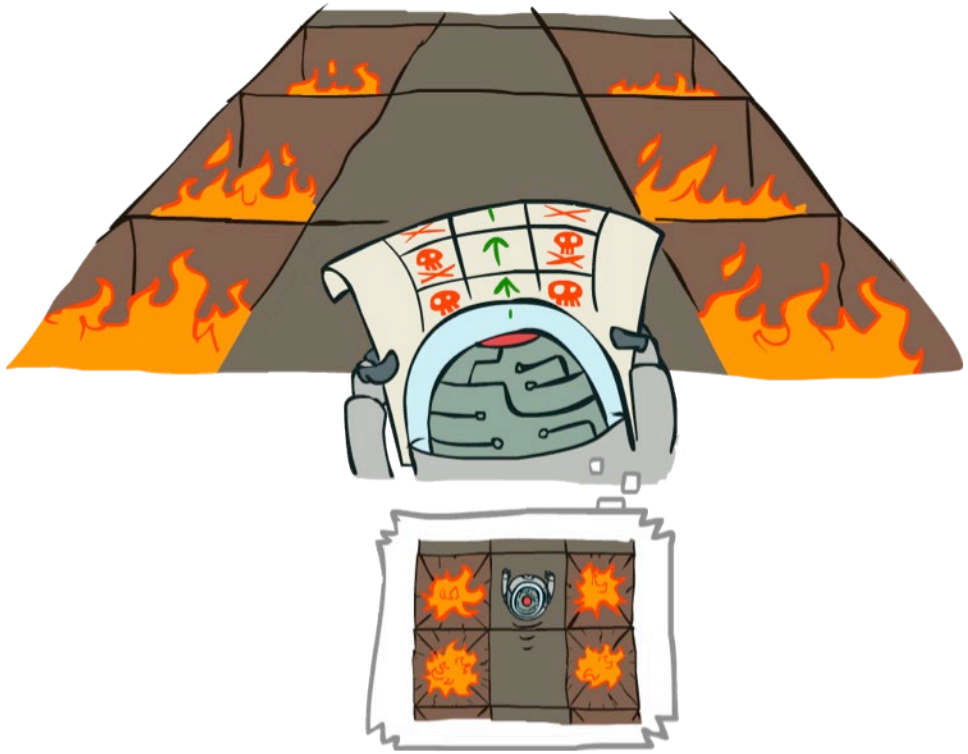
Example: Policy Evaluation

Always Go Right



Bad Policy

Always Go Forward

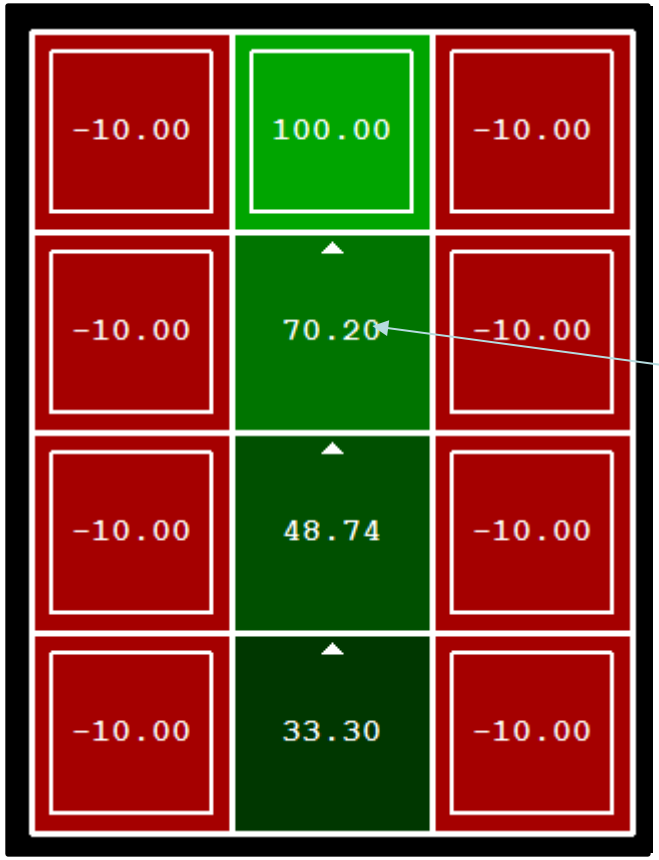


Good Policy

Example: Policy Evaluation

Always Go Right

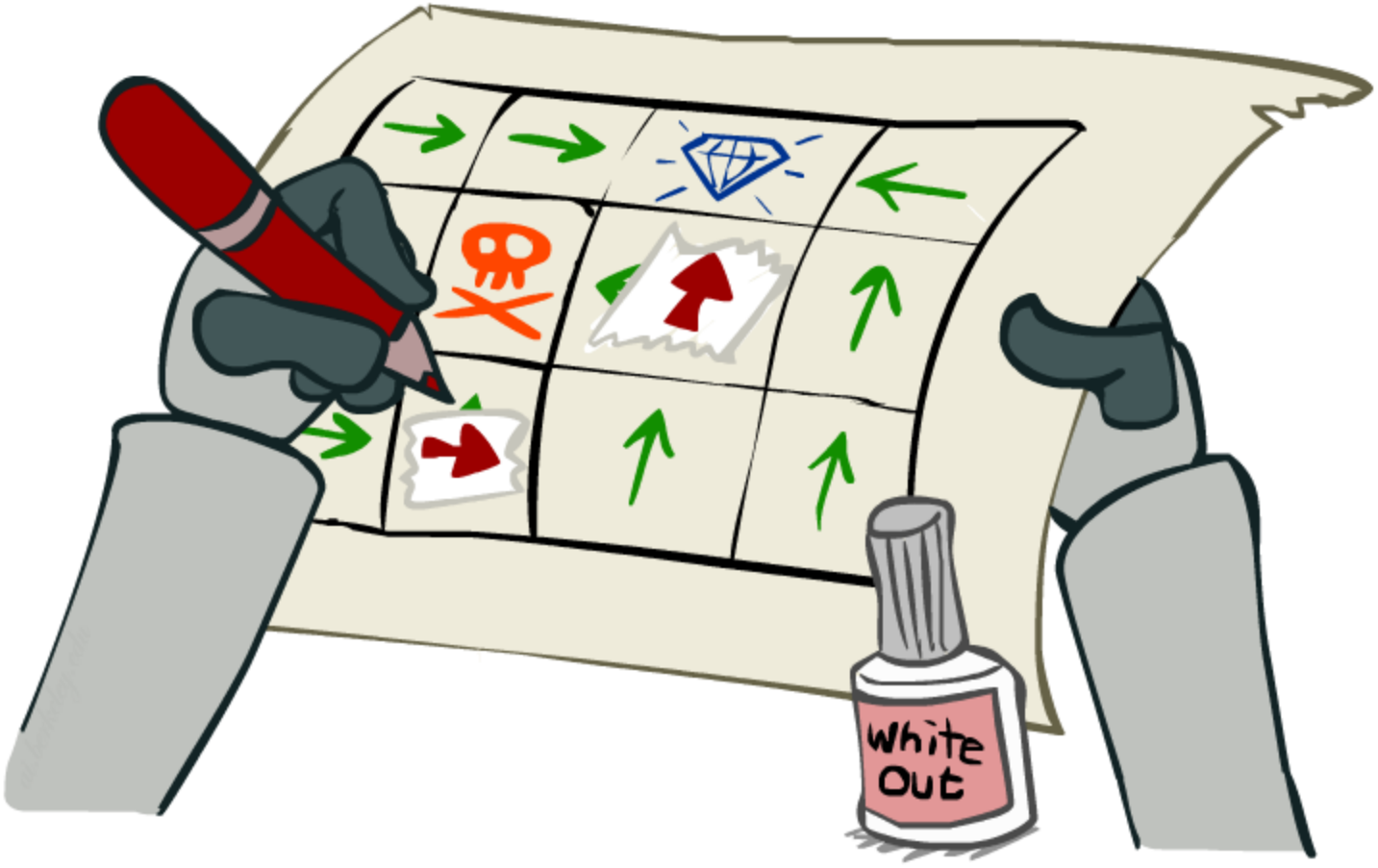
Always Go Forward



Assuming no living reward and Υ=0.9.

$0.8 \times (0.9 \times 100) +$
$0.1 \times (0.9 \times -10) +$
$0.1 \times (0.9 \times -10) = 70.20$

Policy Iteration



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

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

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).

Summary: MDP Equations

- Bellman Equation
- Value iteration equation:

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

- Policy evaluation equation:

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

- Policy improvement equation:

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

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.

- These all look the same!
  - They are all variations of Bellman Equation.
  - They all use one-step lookahead max expected utility (MEU) fragments.
  - They differ only in whether we plug in a fixed policy or max over actions.