Lecture 2: Markov Decision Processes

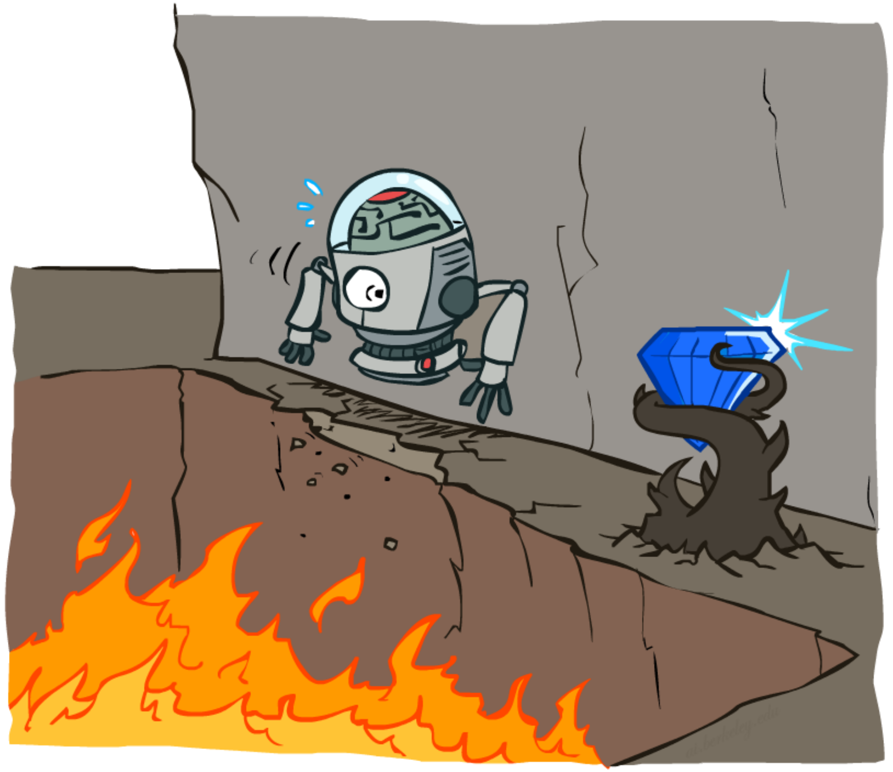
Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

https://shuaili8.github.io

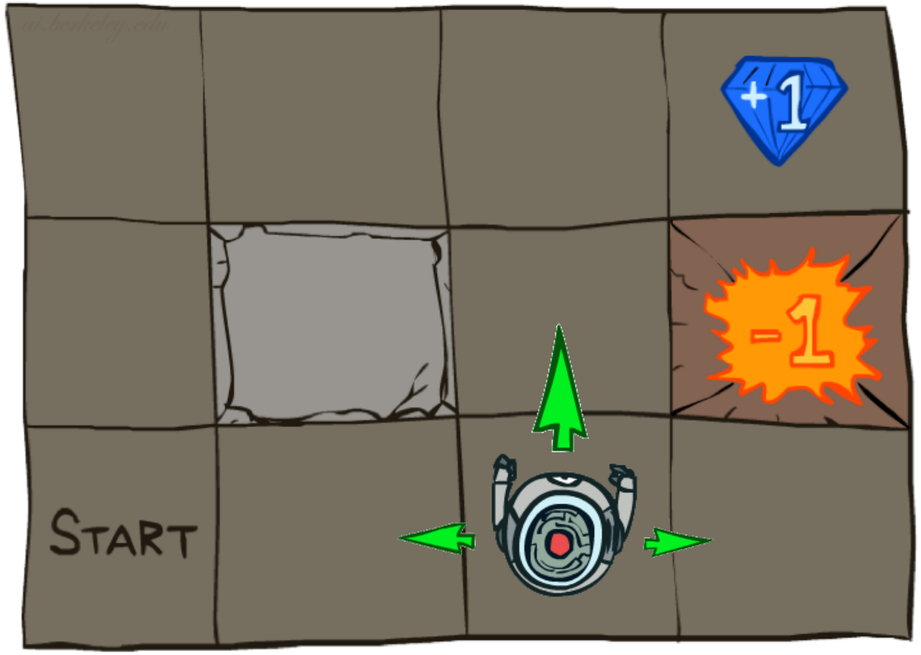
https://shuaili8.github.io/Teaching/AI3601/index.html

Part of slide credits: CMU AI & http://ai.berkeley.edu



Non-Deterministic Search

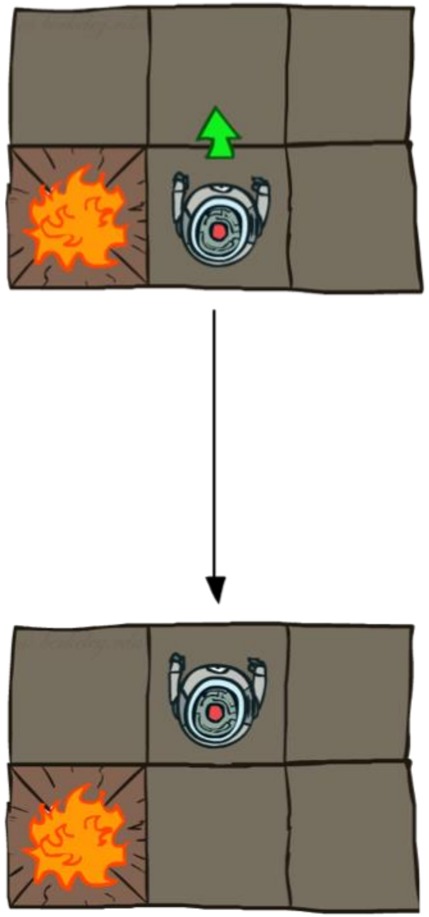
Example: Grid World



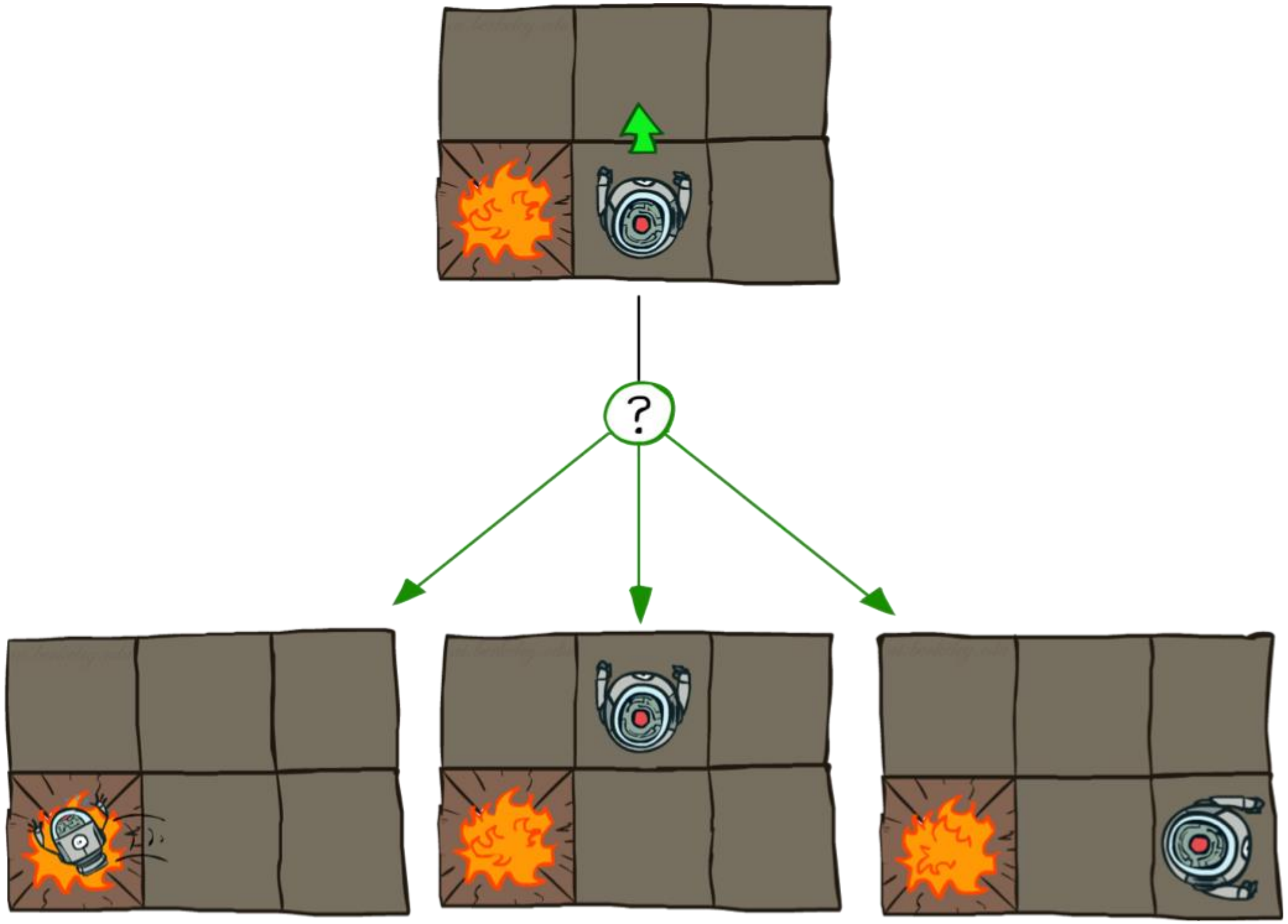
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards

Grid World Actions

Deterministic Grid World



Stochastic Grid World



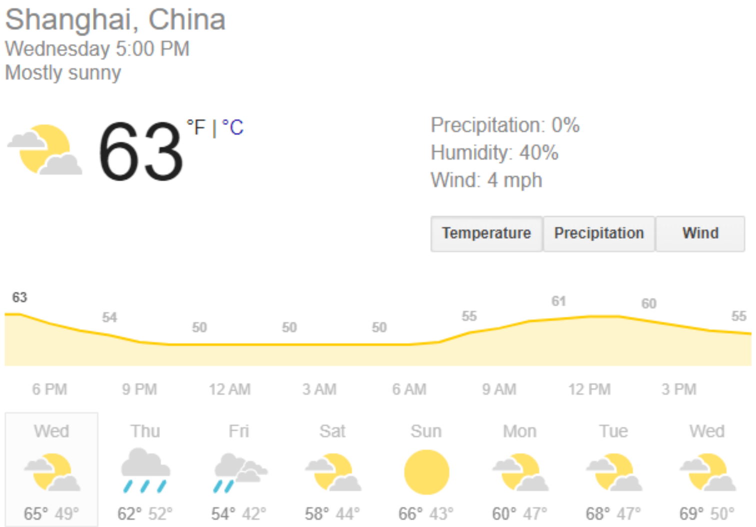
随机过程

- 随机过程是一个或多个事件、随机系统或者随机现象随时间发生演变的过程

$$\mathbb{P}[S_{t+1}|S_1,\dots,S_t]$$

  - 概率论研究静态随机现象的统计规律
  - 随机过程研究动态随机现象的发展规律

布朗运动



Shanghai, China
Wednesday 5:00 PM
Mostly sunny

63 °F | °C

Precipitation: 0%
Humidity: 40%
Wind: 4 mph

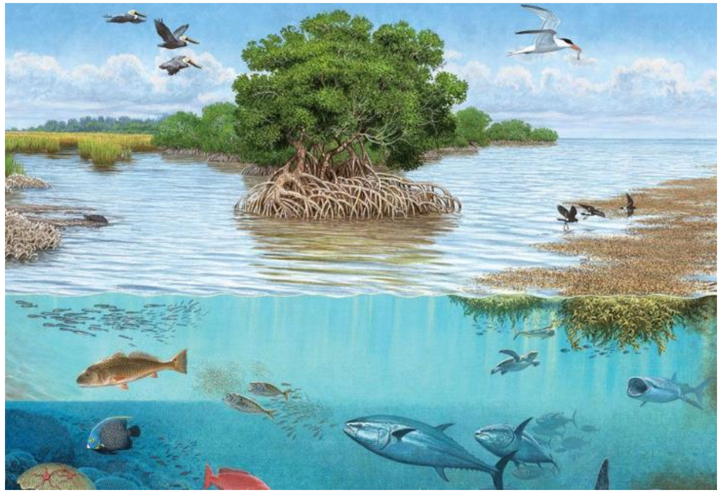
Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed
65° 49°	62° 52°	54° 42°	58° 44°	66° 43°	60° 47°	68° 47°	69° 50°

天气变化

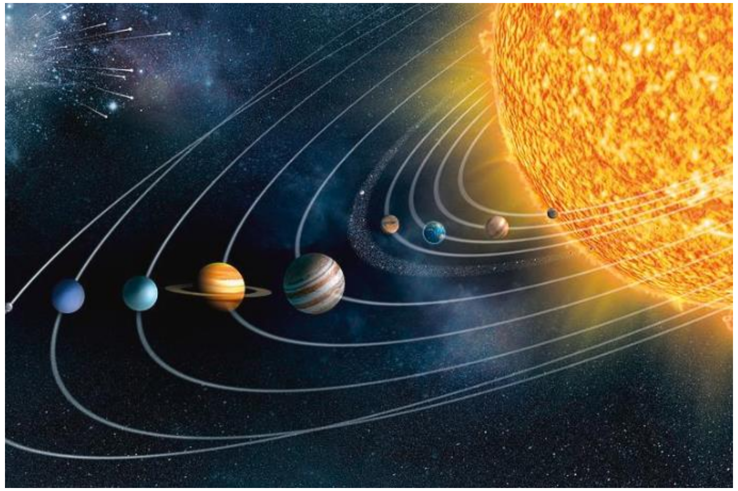
随机过程

足球比赛

城市交通



生态系统



星系

马尔可夫过程

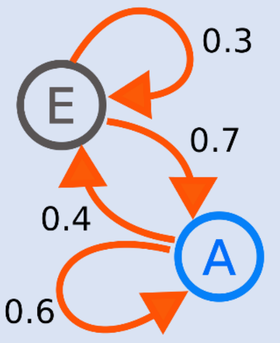
- 马尔可夫过程（Markov Process）是具有马尔可夫性质的随机过程

  “The future is independent of the past given the present”

- 定义：
  - 状态S_t是马尔可夫的，当且仅当

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \dots, S_t]$$

- 性质：
  - 状态从历史（history）中捕获了所有相关信息
  - 当状态已知的时候，可以抛开历史不管
  - 也就是说，当前状态是未来的充分统计量



一个具有两个转换状态的马尔可夫链

马尔可夫决策过程

- 马尔可夫决策过程（Markov Decision Process，MDP）
  - 提供了一套为在结果部分随机、部分在决策者的控制下的决策过程建模的数学框架

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \dots, S_t]$$

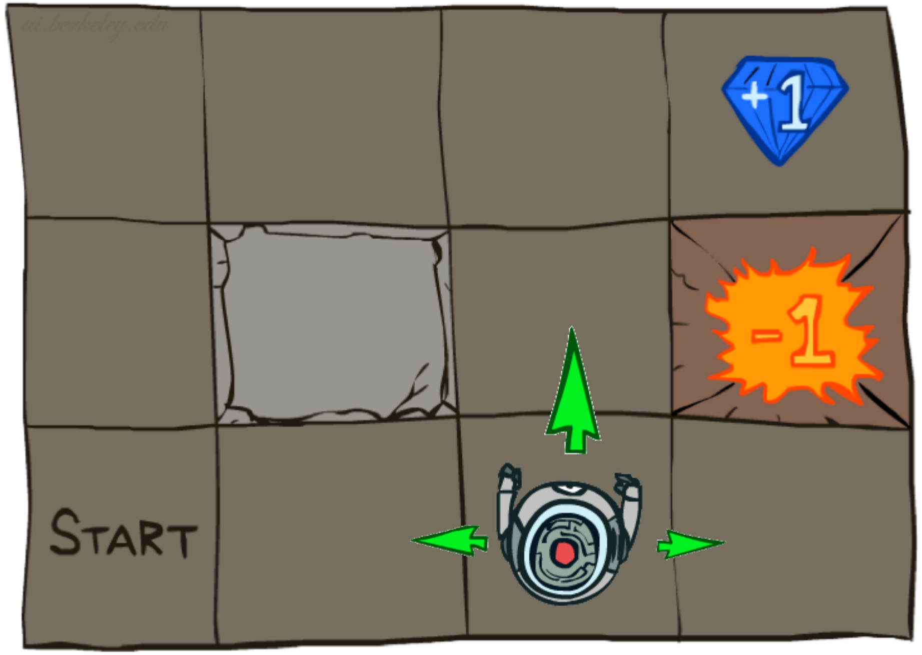
$$\mathbb{P}[S_{t+1}|S_t, A_t]$$

- MDP形式化地描述了一种强化学习的环境
  - 环境完全可观测
  - 即，当前状态可以完全表征过程（马尔可夫性质）

Markov Decision Processes

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

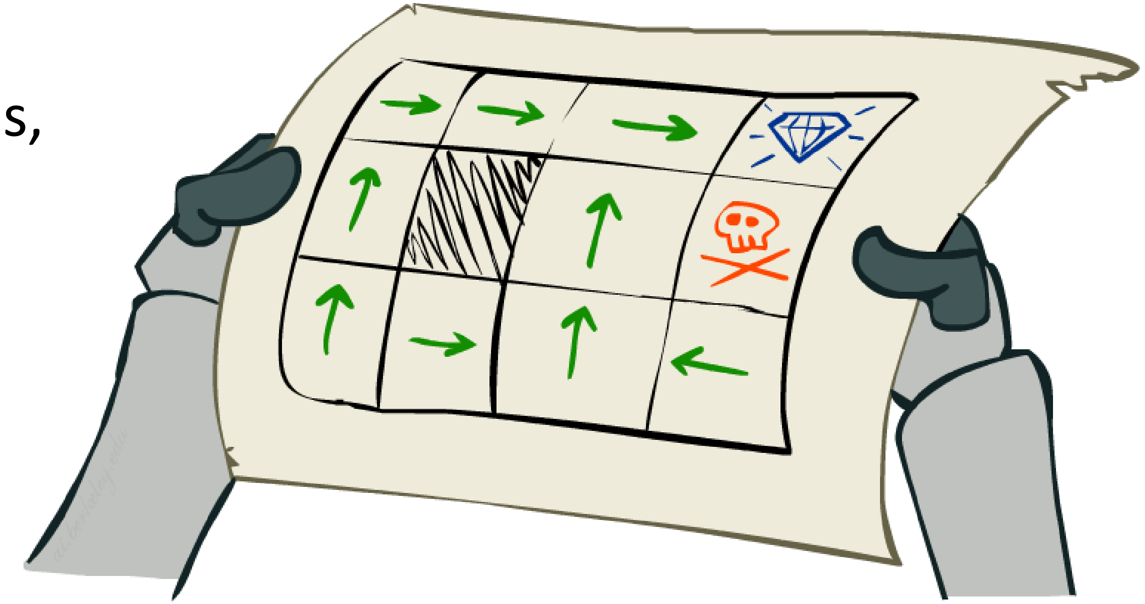
Policies

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

- For MDPs, we want an optimal

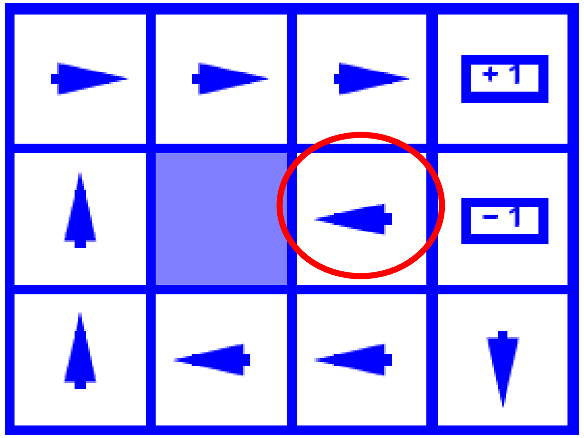
  policy $\pi^*: S \rightarrow A$

  - 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

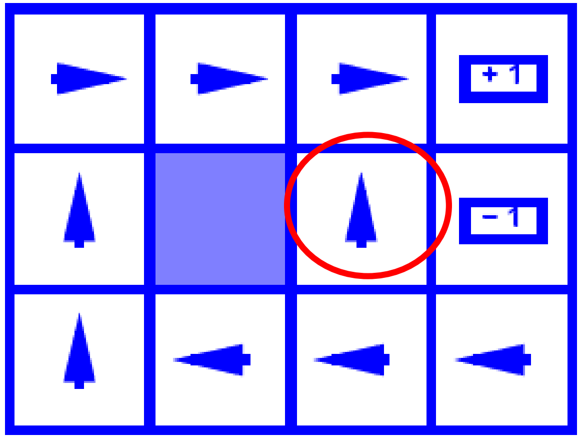


Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

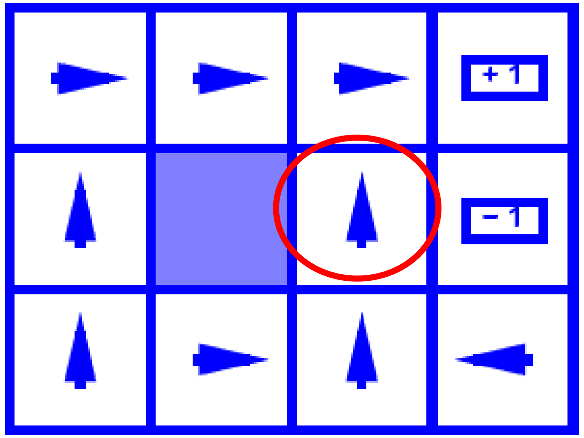
Optimal Policies



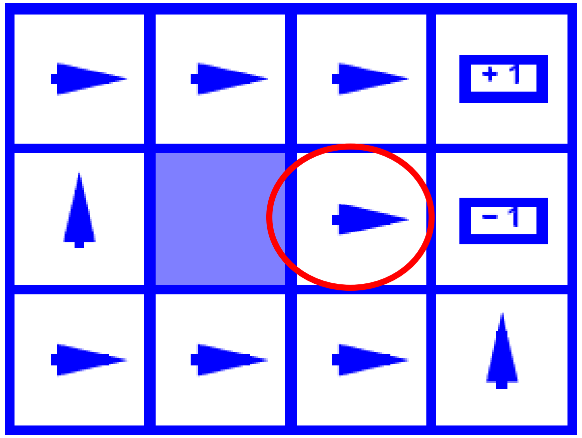
R(s) = -0.01



R(s) = -0.03

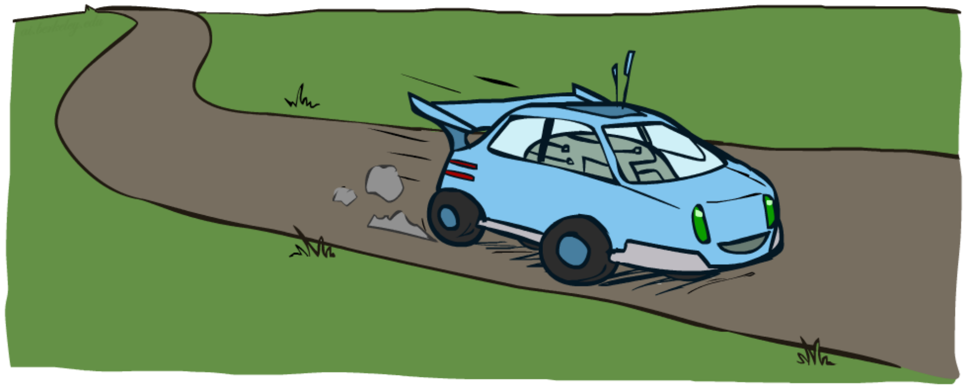


R(s) = -0.4

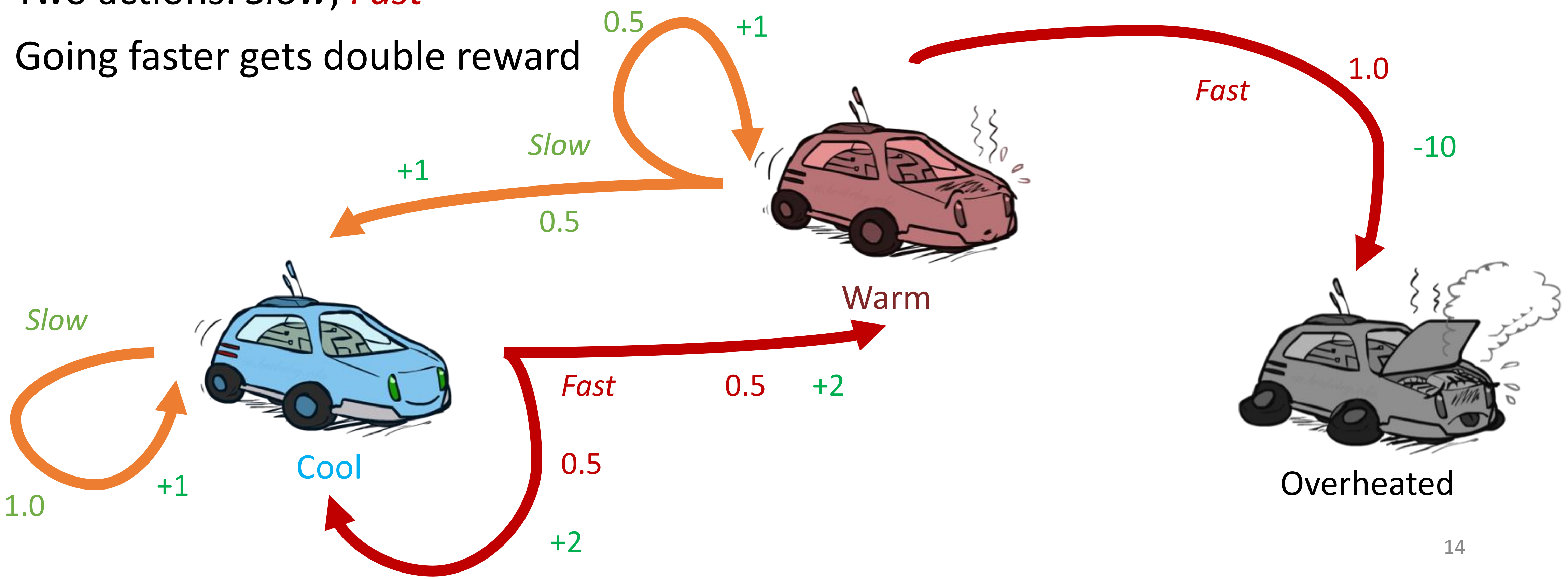


R(s) = -2.0

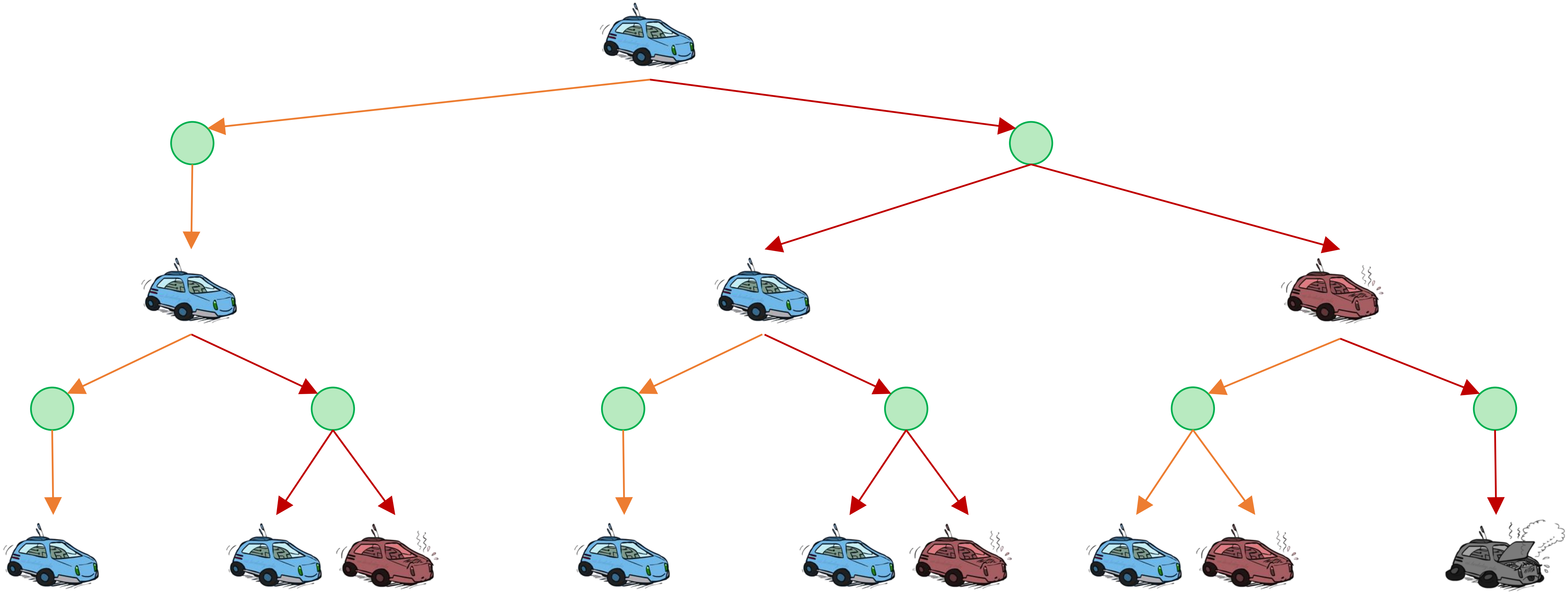
Example: Racing



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

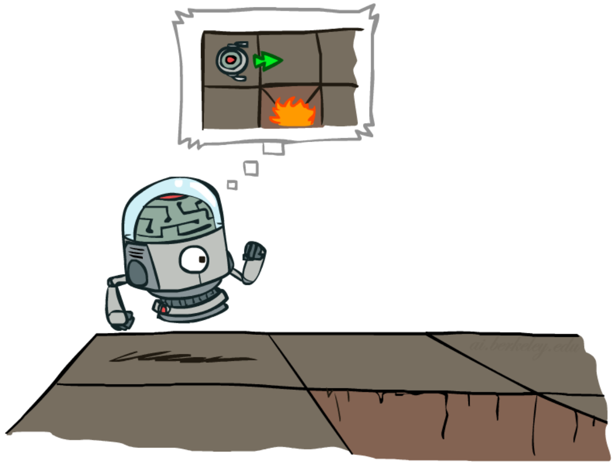


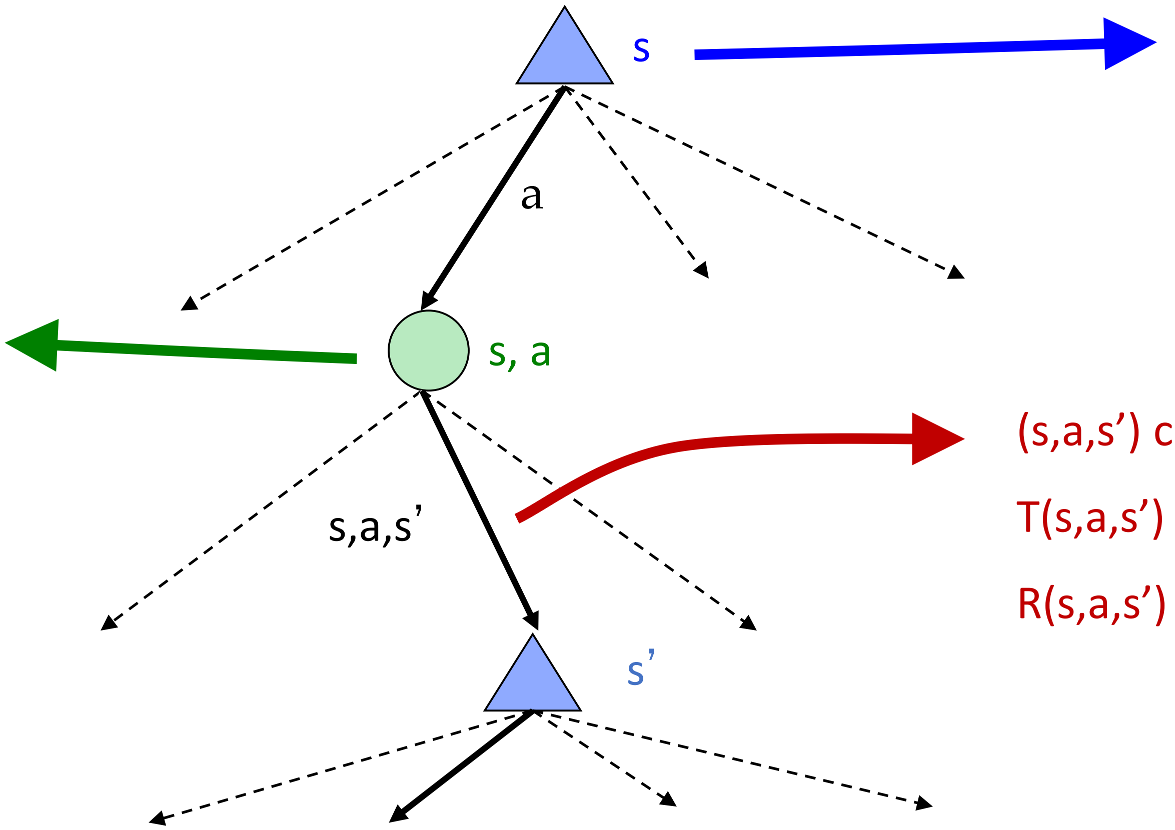
Example: Racing - Search Tree



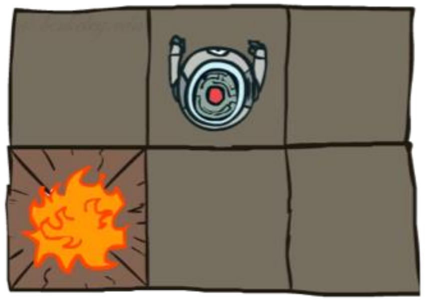
MDP Search Trees

- Each MDP state projects an expectimax-like search tree





s is a *state*



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

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

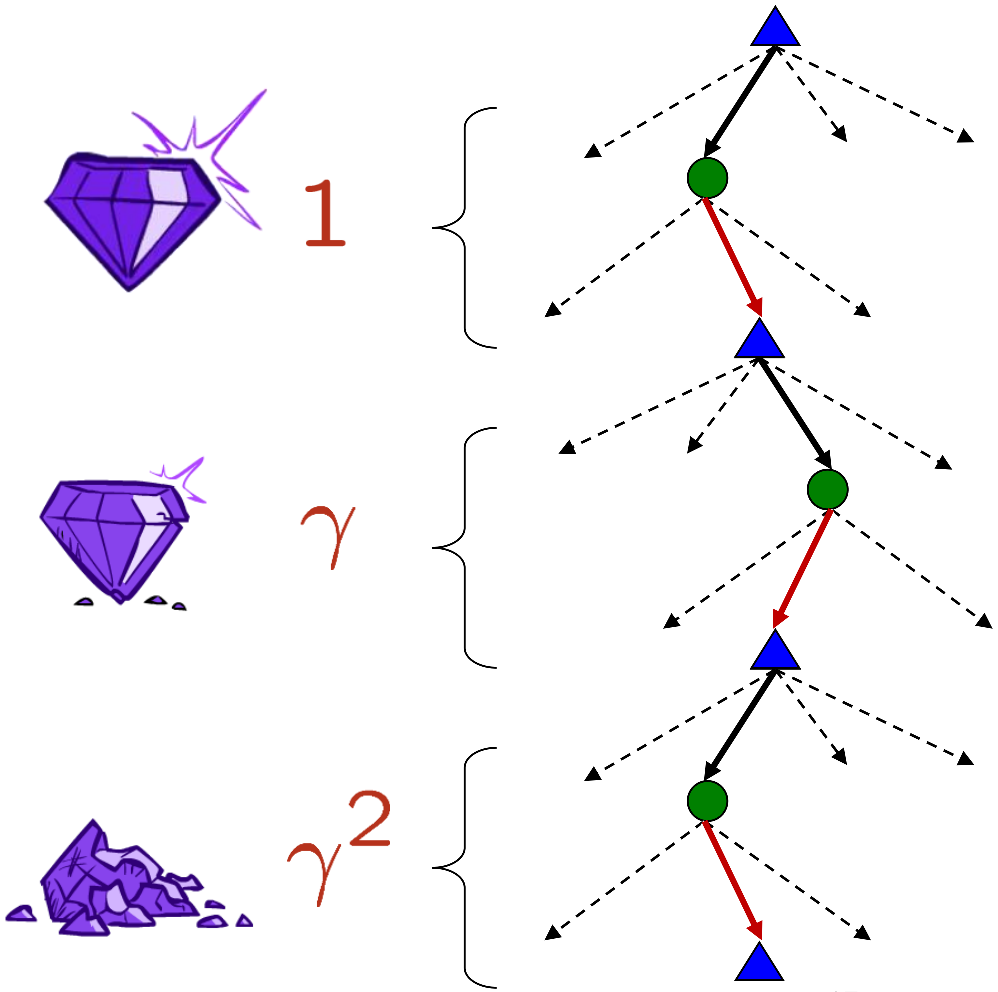
R(s,a,s')

Utilities of Sequences: Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once

- Why discount?
  - Reward now is better than later
  - Can also think of it as a 1-gamma chance of ending the process at every step
  - Also helps our algorithms converge

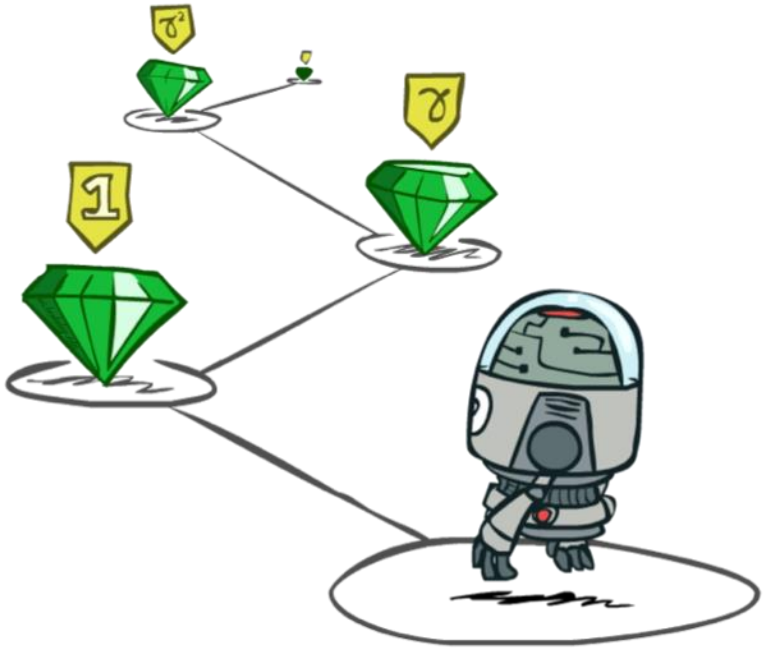
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



Utilities of Sequences: Stationary Preferences

- Theorem: if we assume stationary preferences:

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



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

Counterexample

- Can $U_{\gamma} + U_{\gamma'}$ define a stationary preference?

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

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

- No!
- Example:
  - $(U_{0.9} + U_{0.5})\left(\frac{3}{4}, 0, 0, \ldots\right) > (U_{0.9} + U_{0.5})(0, 1, 0, \ldots)$
  - $(U_{0.9} + U_{0.5})\left(r, \frac{3}{4}, 0, \ldots\right) < (U_{0.9} + U_{0.5})(r, 0, 1, 0, 0, \ldots)$

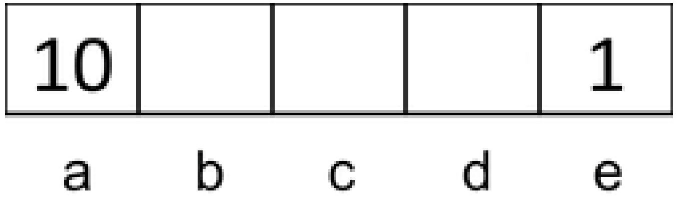
Proof Sketch

- Theorem (F. Riesz) For any inner product space H, let f be a continuous linear functional, that is $f: H \to \mathbb{R}$ is continuous and satisfies $f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$. Then f can be written as
$$f(x) =< z, x >$$
for some $z \in H$

- Then by $a_0 + \gamma_1 a_1 > b_0 + \gamma_1 b_1 \Leftrightarrow \gamma_1 a_0 + \gamma_2 a_1 > \gamma_1 b_0 + \gamma_2 b_1$ which says $(a_0 - b_0) + \gamma_1(a_1 - b_1) > 0 \Leftrightarrow \gamma_1(a_0 - b_0) + \gamma_2(a_1 - b_1) > 0$
Then there must have $\gamma_2 = \gamma_1^2$
- Similarly for the rest

Quiz: Discounting

- Given:



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

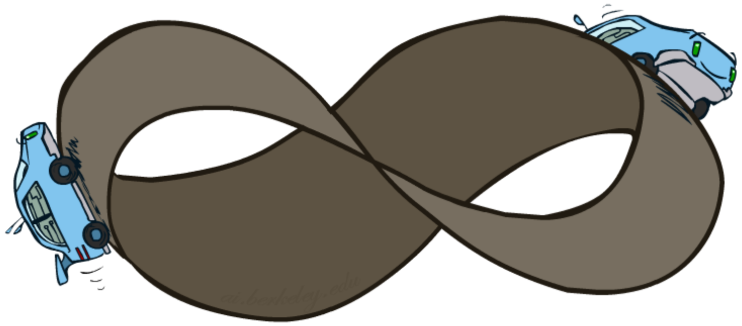
- Quiz 1: For $\gamma = 1$, what is the optimal policy?

- 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?
- 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)



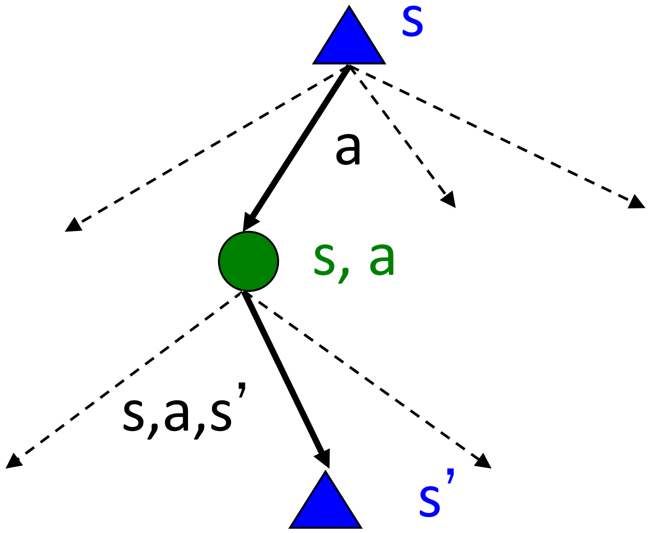
    - Discounting: use 0 < γ < 1

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

        - 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

MDP的动态

- MDP的动态如下所示：
  - 从状态s_0开始
  - 智能体选择某个动作$a_0 \in A$
  - 智能体得到奖励$R(s_0, a_0)$
  - MDP随机转移到下一个状态$s_1 \sim P_{s_0,a_0}$
  - 这个过程不断进行

$$s_0 \xrightarrow{a_0,R(s_0,a_0)} s_1 \xrightarrow{a_1,R(s_1,a_1)} s_2 \xrightarrow{a_2,R(s_2,a_2)} s_3 \quad \cdots$$

  - 直到终止状态s_T出现为止，或者无止尽地进行下去
  - 智能体的总回报为

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ \cdots$$

MDP的动态性

- 在许多情况下，奖励只和状态相关
  - 比如，在迷宫游戏中，奖励只和位置相关
  - 在围棋中，奖励只基于最终所围地盘的大小有关
- 这时，奖励函数为 $R(s): S \longmapsto \mathbb{R}$
- MDP的过程为

$$s_0 \xrightarrow{a_0, R(s_0)} s_1 \xrightarrow{a_1, R(s_1)} s_2 \xrightarrow{a_2, R(s_2)} s_3 \quad \cdots$$

- 累积奖励为

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

REVIEW: 在与动态环境的交互中学习

有监督、无监督学习

Model ←

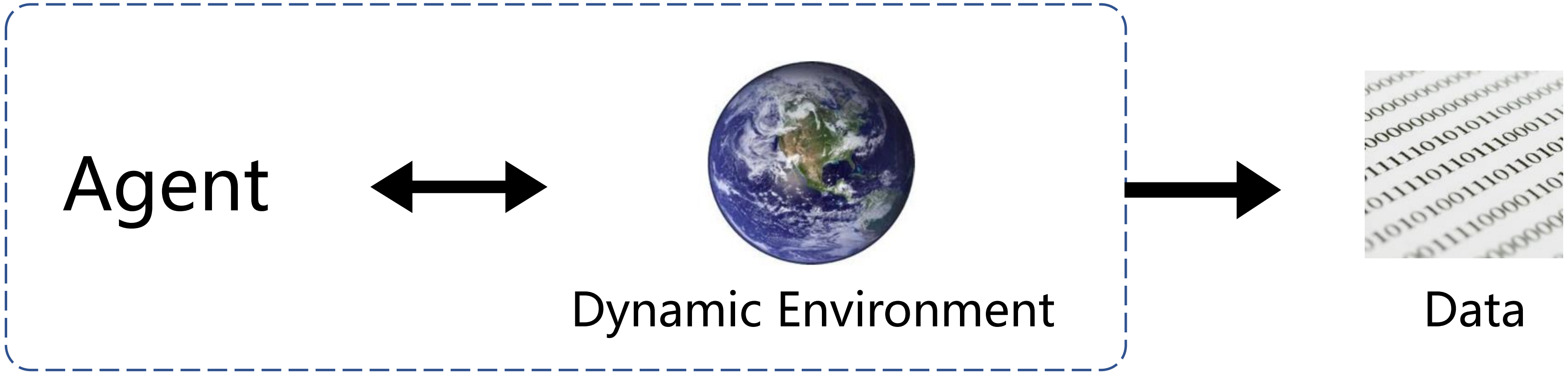
Fixed Data

强化学习

Agent ↔

Dynamic Environment

和动态环境交互产生的数据分布



Agent

Dynamic Environment

Data

- 给定同一个动态环境（即MDP），不同的策略采样出来的(状态-行动)对的分布是不同的

- 占用度量（Occupancy Measure）

$$\rho^{\pi}(s,a) = \sum_{t=0}^{T} \gamma^{t}\, \mathbb{P}(s_t = s, a_t = a | s_0, \pi)$$

占用度量和策略

- 占用度量（Occupancy Measure）

$$\rho^{\pi}(s,a) = \sum_{t=0}^{T} \gamma^{t}\, \mathbb{P}(s_t = s, a_t = a | s_0, \pi)$$

- 定理1：和同一个动态环境交互的两个策略π_1和π_2得到的占用度量ρ^{π_1}和ρ^{π_2}满足

$$\rho^{\pi_1} = \rho^{\pi_2} \text{ iff } \pi_1 = \pi_2$$

- 定理2：给定一占用度量ρ，可生成该占用度量的唯一策略是

$$\pi_{\rho}(a|s) = \frac{\rho(s,a)}{\sum_{a'} \rho(s,a')}$$

U. Syed, M. Bowling, and R. E. Schapire. Apprenticeship learning using linear programming. ICML 2008.

占用度量和策略

- 占用度量（Occupancy Measure)

$$\rho^{\pi}(s,a) = \sum_{t=0}^{T} \gamma^t \, \mathbb{P}(s_t = s, a_t = a | s_0, \pi)$$

- 状态占用度量

$$\rho^{\pi}(s) = \sum_{t=0}^{T} \gamma^t \, \mathbb{P}(s_t = s | s_0, \pi)$$

$$= \sum_{t=0}^{T} \gamma^t \, \mathbb{P}(s_t = s | s_0, \pi) \sum_{a'} \pi(a_t = a | s_t = s)$$

$$= \sum_{a} \sum_{t=0}^{T} \gamma^t \, \mathbb{P}(s_t = s, a_t = a | s_0, \pi)$$

$$= \sum_{a} \rho^{\pi}(s,a)$$

U. Syed, M. Bowling, and R. E. Schapire. Apprenticeship learning using linear programming. ICML 2008.

占用度量和累计奖励

- 占用度量（Occupancy Measure)

$$\rho^{\pi}(s,a) = \sum_{t=0}^{T} \gamma^t \, \mathbb{P}(s_t = s, a_t = a | s_0, \pi)$$

□ 策略的累积奖励为

$$V(\pi) = \mathbb{E}_{(s_0,a_0,s_1,a_1,\dots)\text{is a trajectory}}[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \ \cdots]$$

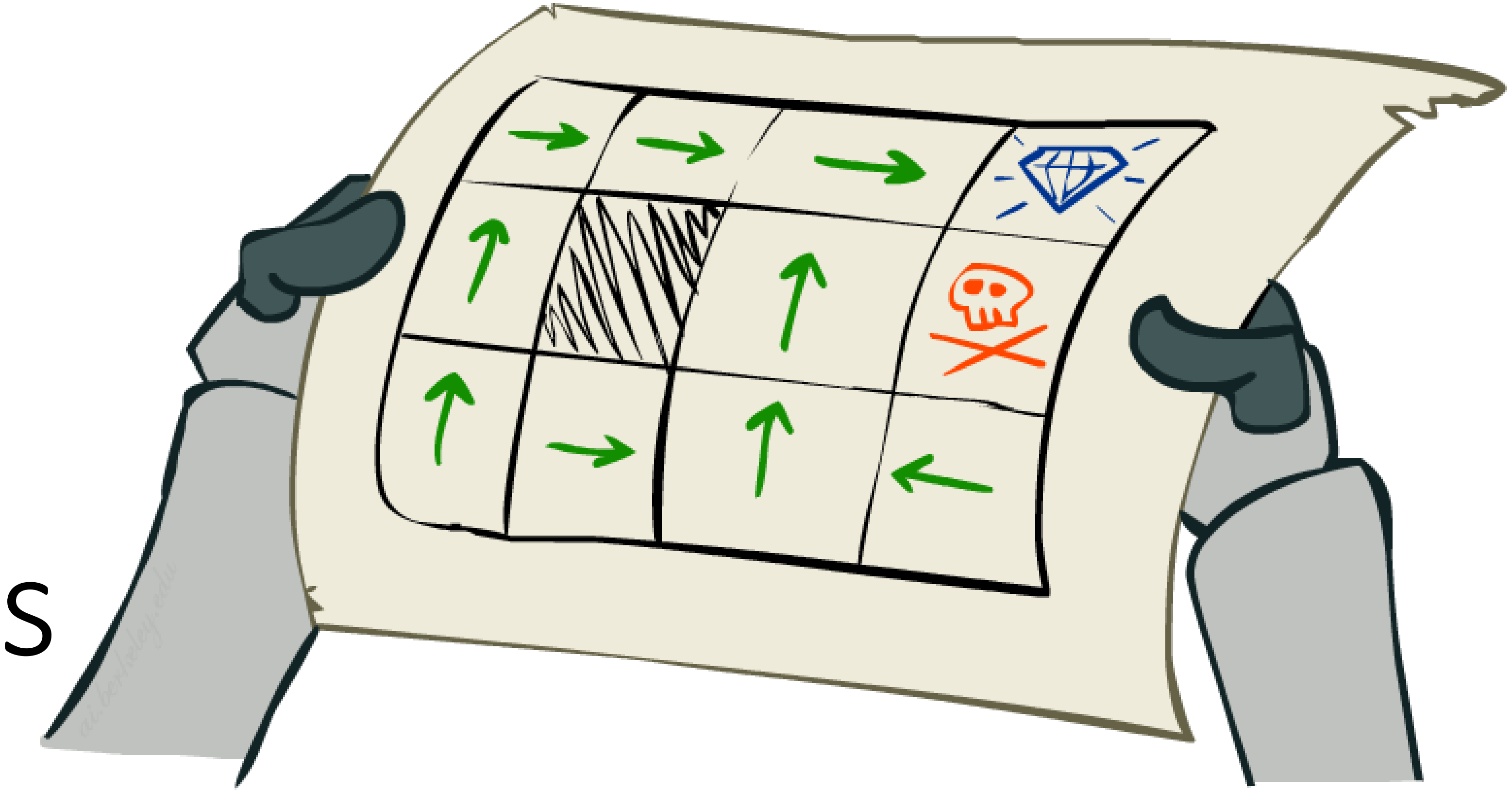
$$= \sum_{s,a} \left[\sum_{t=0}^{T} \gamma^t \mathbb{P}(s_t = s, a_t = a | s_0, \pi) \right] R(s,a)$$

$$= \sum_{s,a} \rho^{\pi}(s,a) R(s,a) \ = \ \mathbb{E}_{\pi}[R(s,a)]$$

强化学习中的简写

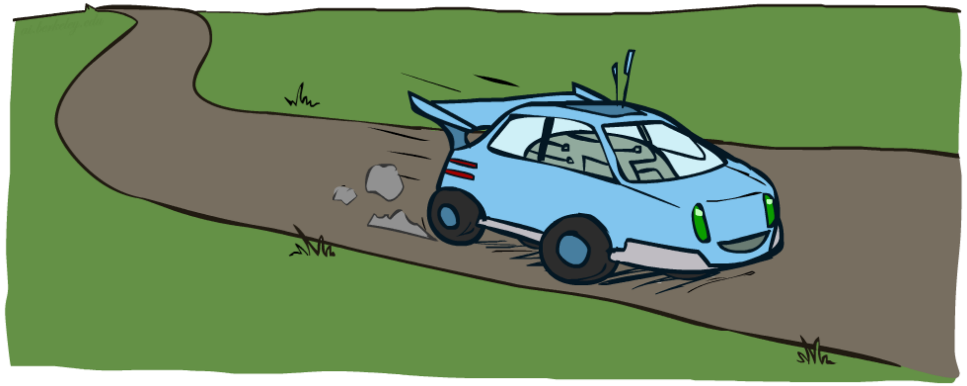
U. Syed, M. Bowling, and R. E. Schapire. Apprenticeship learning using linear programming. ICML 2008.

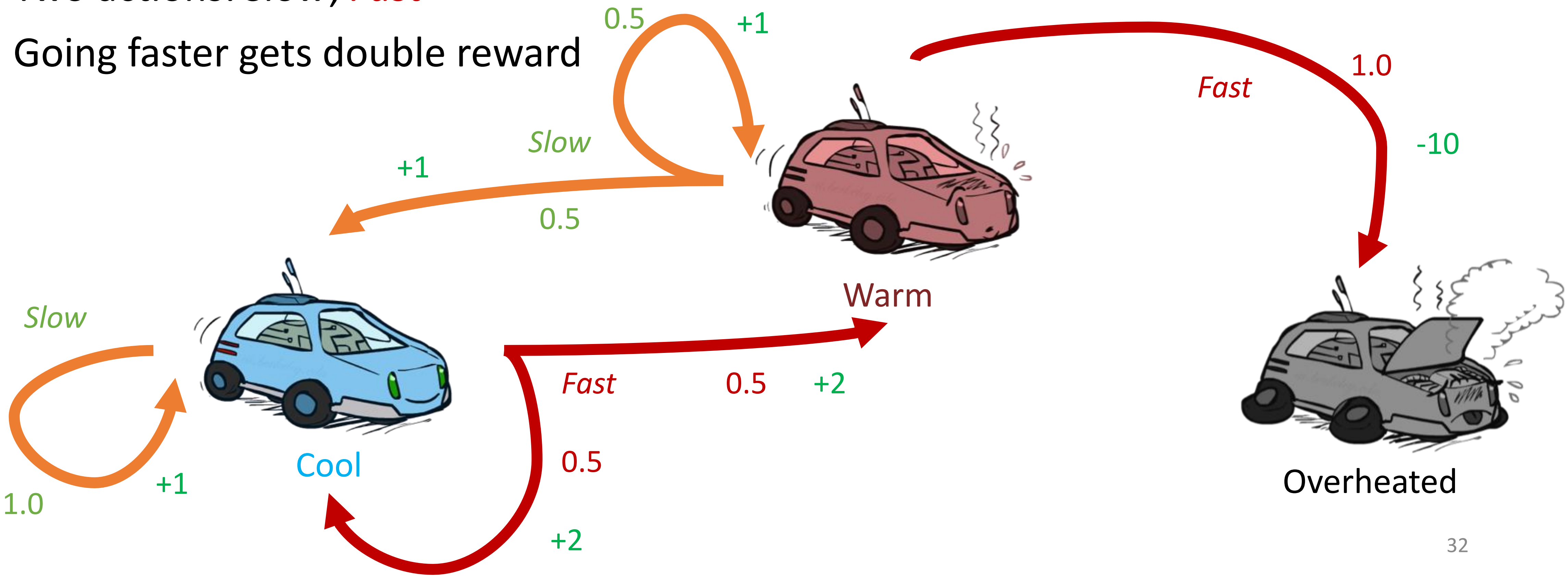
Solving MDPs



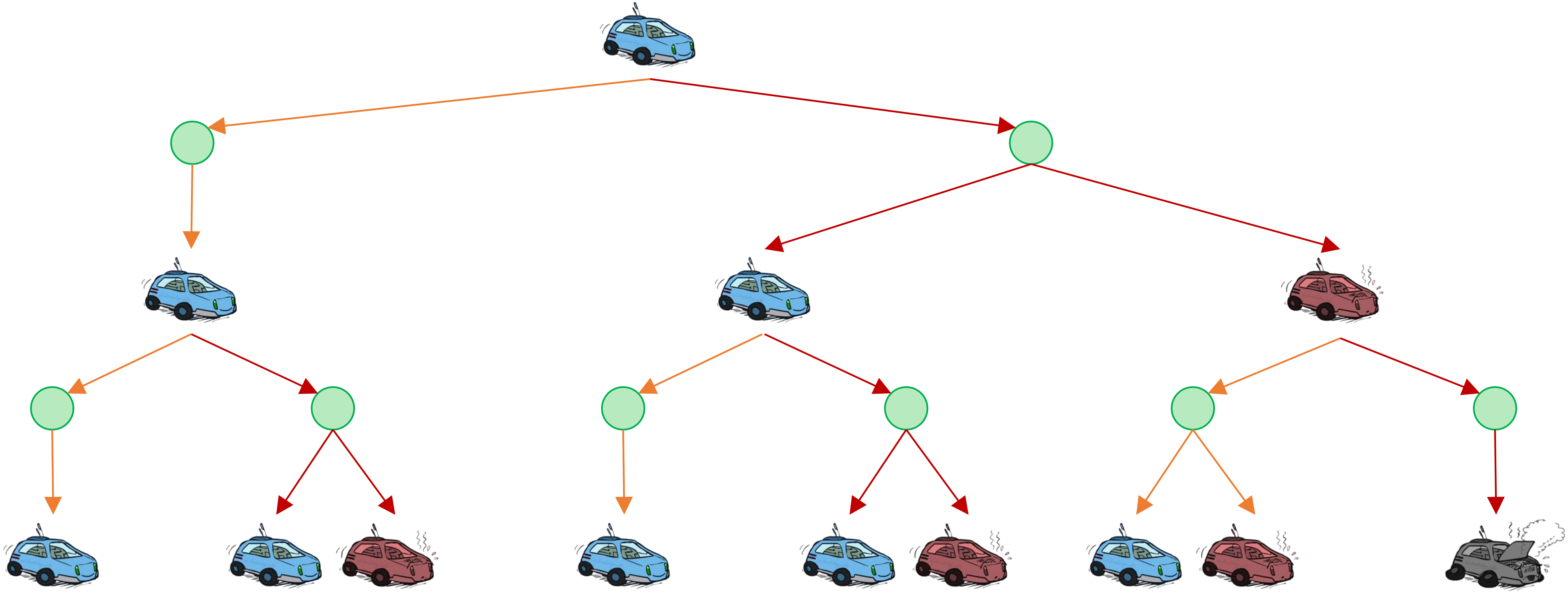
Racing MDP

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

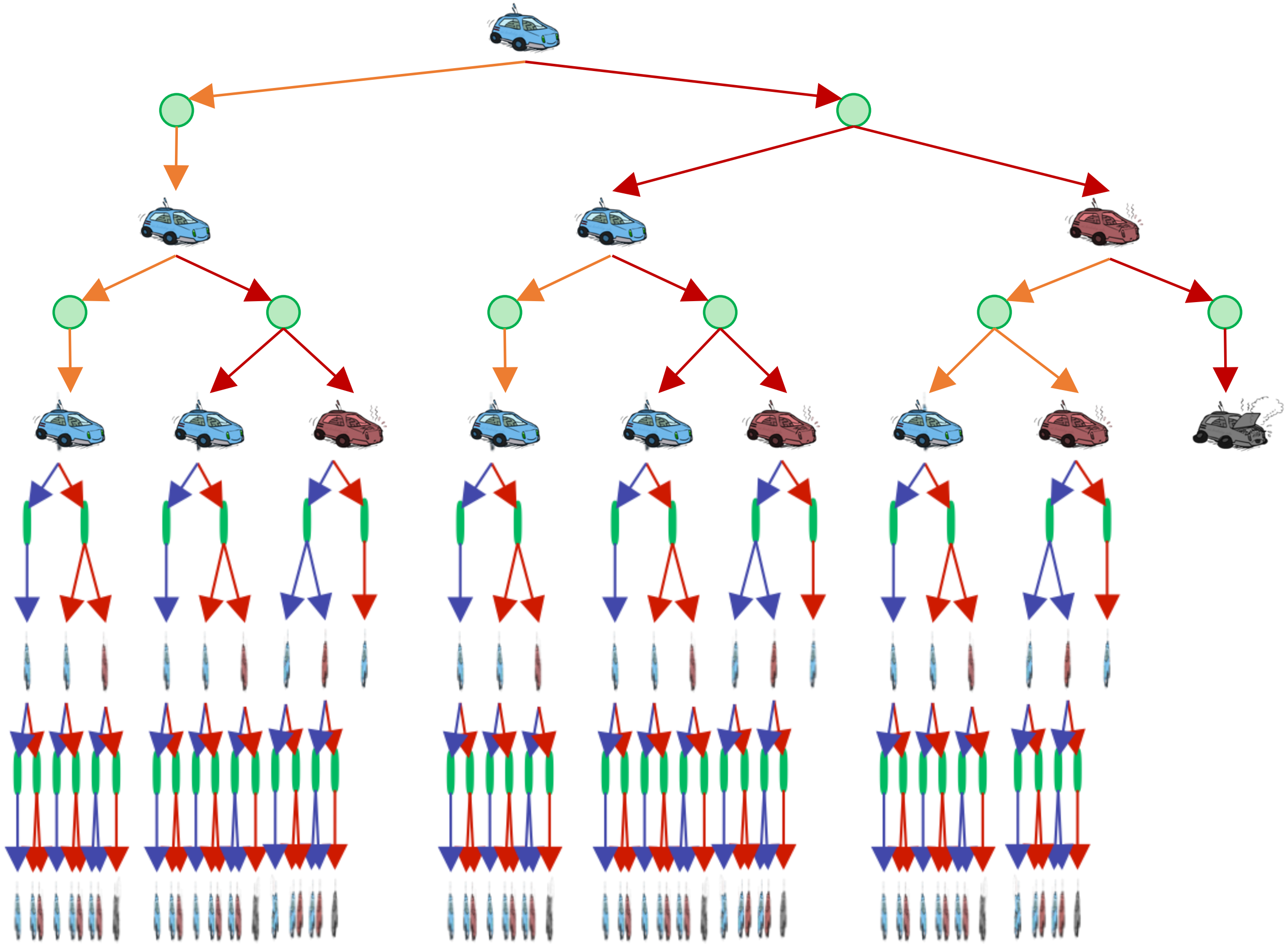




Racing Search Tree

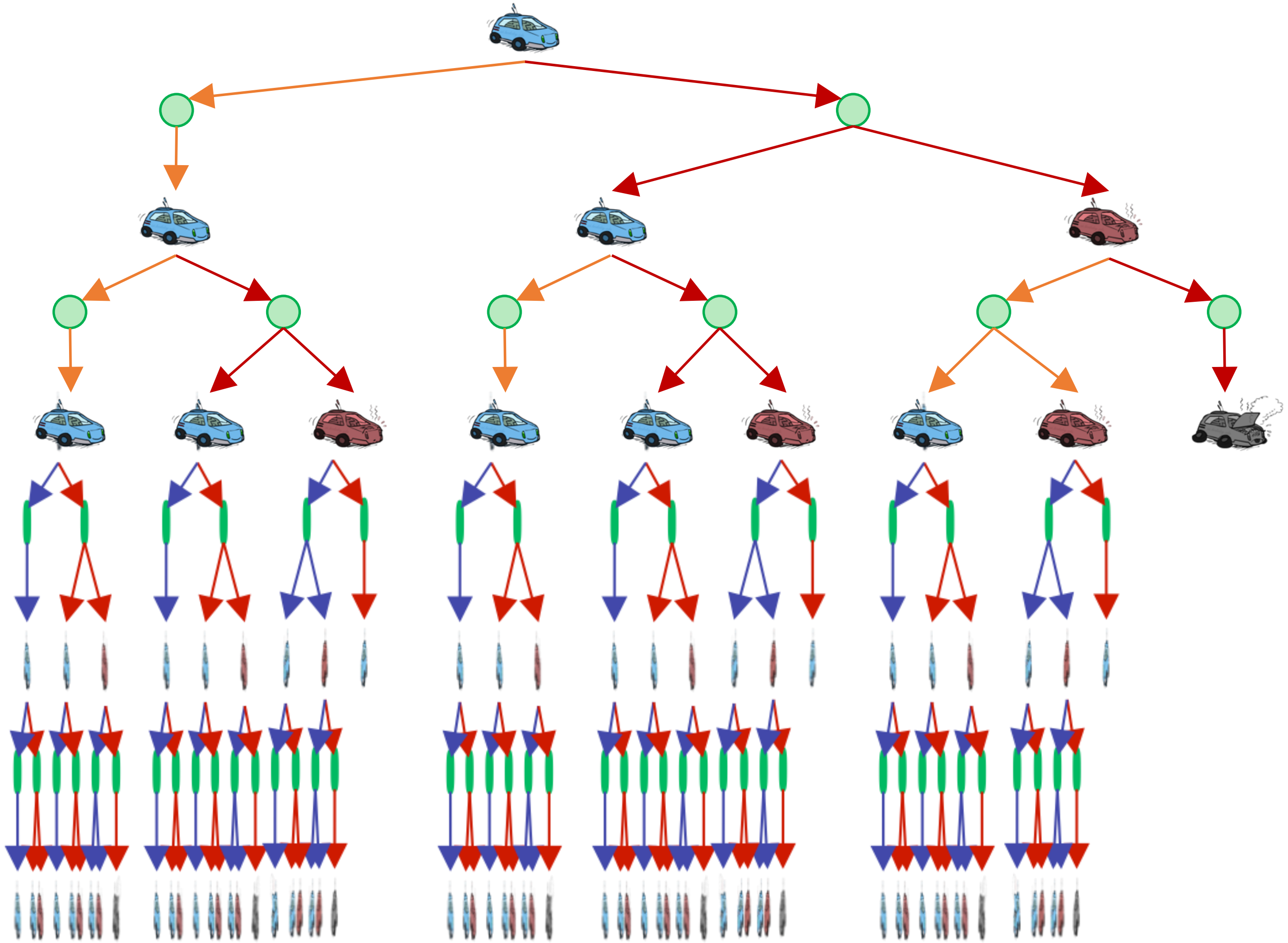


Racing Search Tree 2



Racing Search Tree 3

- 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 $\gamma < 1$

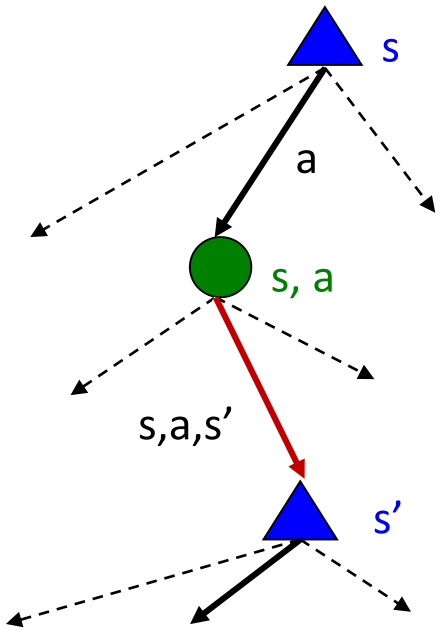


Optimal Quantities

- 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:
  - π*(s) = optimal action from state s

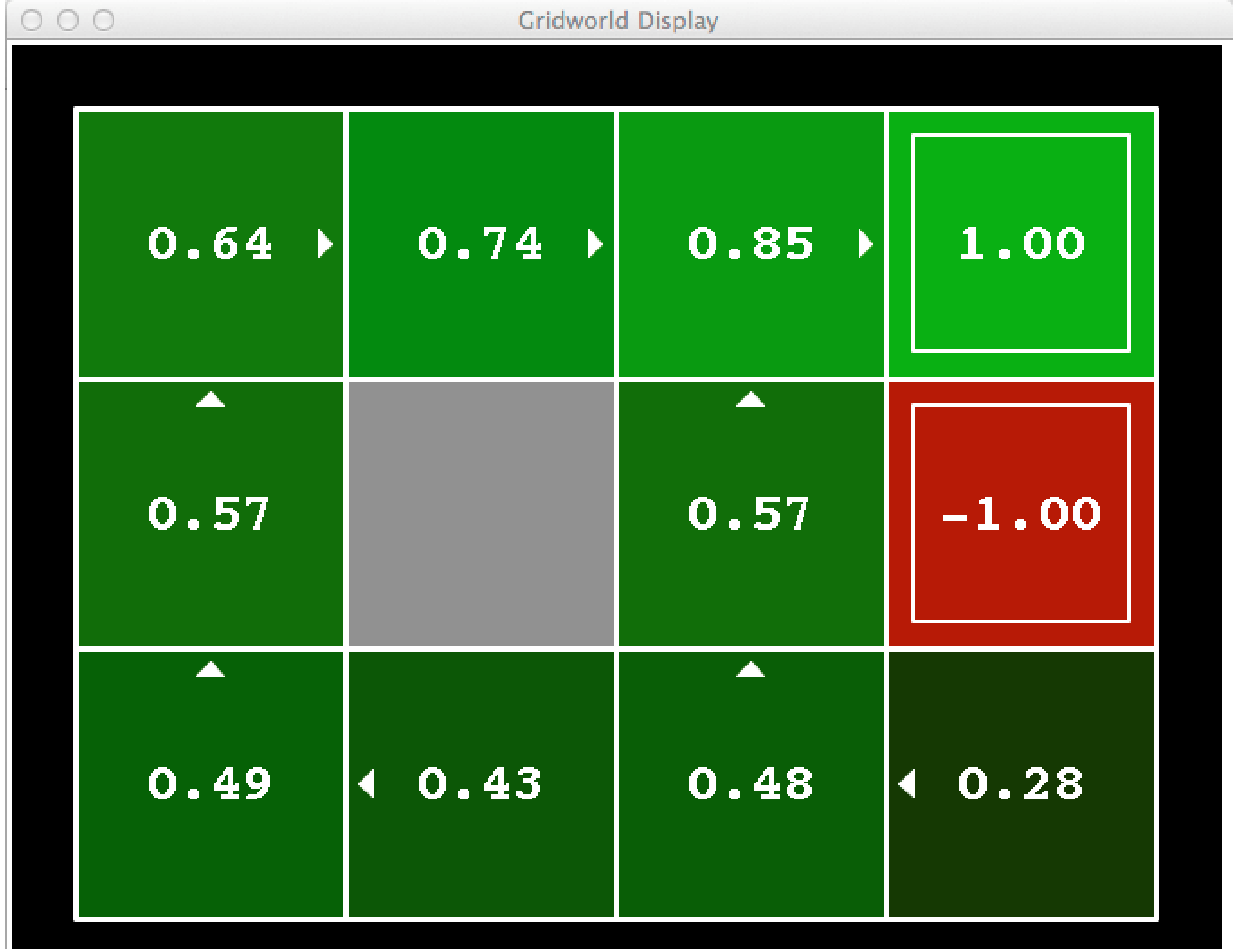


s is a *state*

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

(s,a,s’) is a *transition*

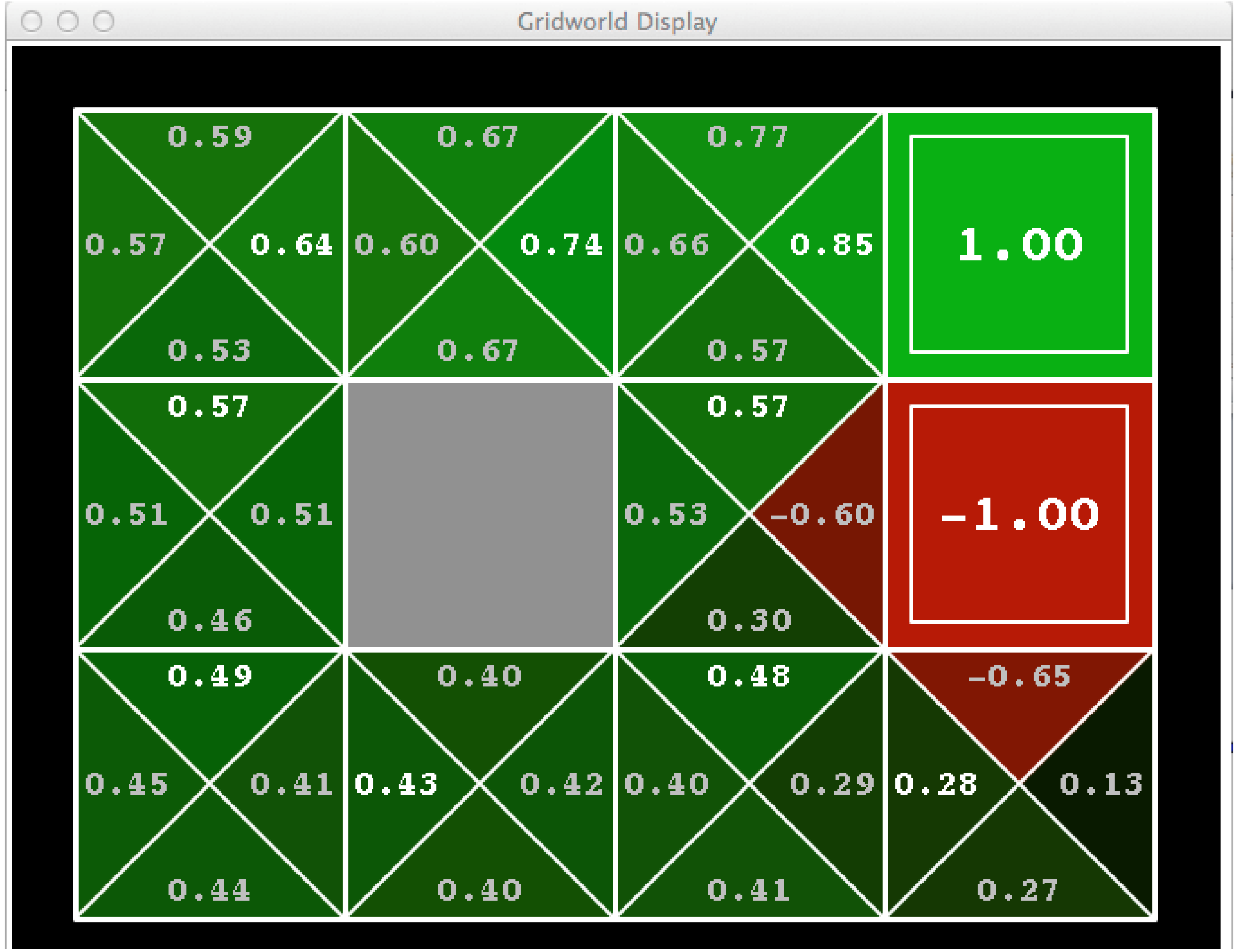
Gridworld V* Values



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

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld Q* Values



0.59 0.57 0.64 0.53	0.67 0.60 0.74 0.67	0.77 0.66 0.85 0.57	1.00
0.57 0.51 0.51 0.46		0.57 0.53 -0.60 0.30	-1.00
0.49 0.45 0.41 0.44	0.40 0.43 0.42 0.40	0.48 0.40 0.29 0.41	-0.65 0.28 0.13 0.27

Noise = 0.2
Discount = 0.9
Living reward = 0

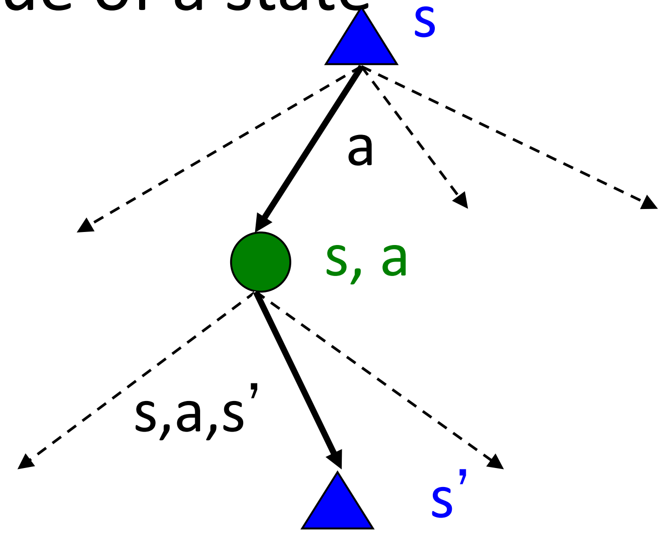
Values of States

- 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:

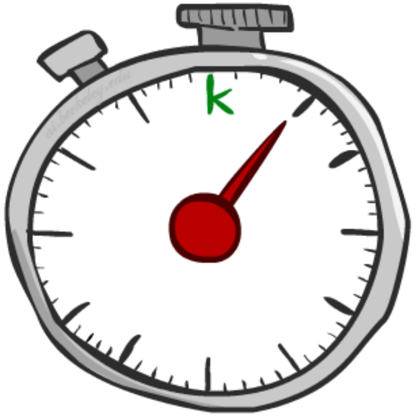
$$V^*(s) = \max_a Q^*(s,a)$$

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

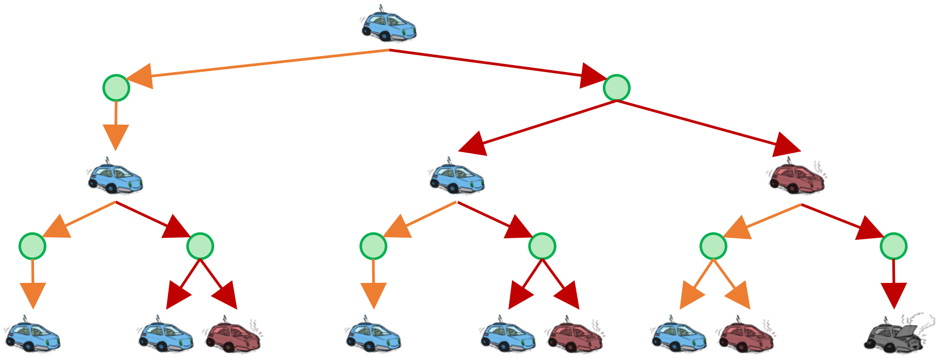
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

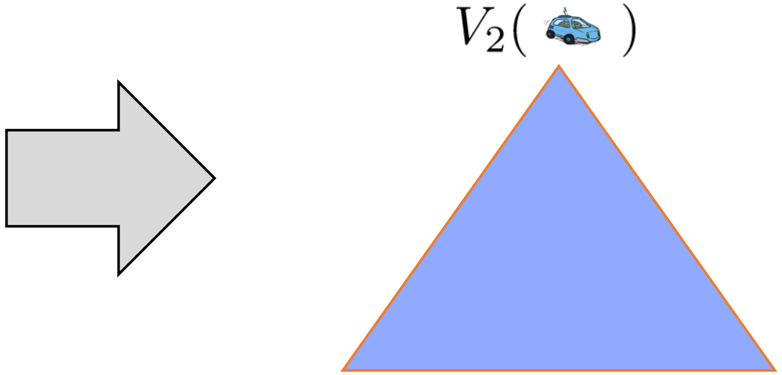


Time-Limited Values



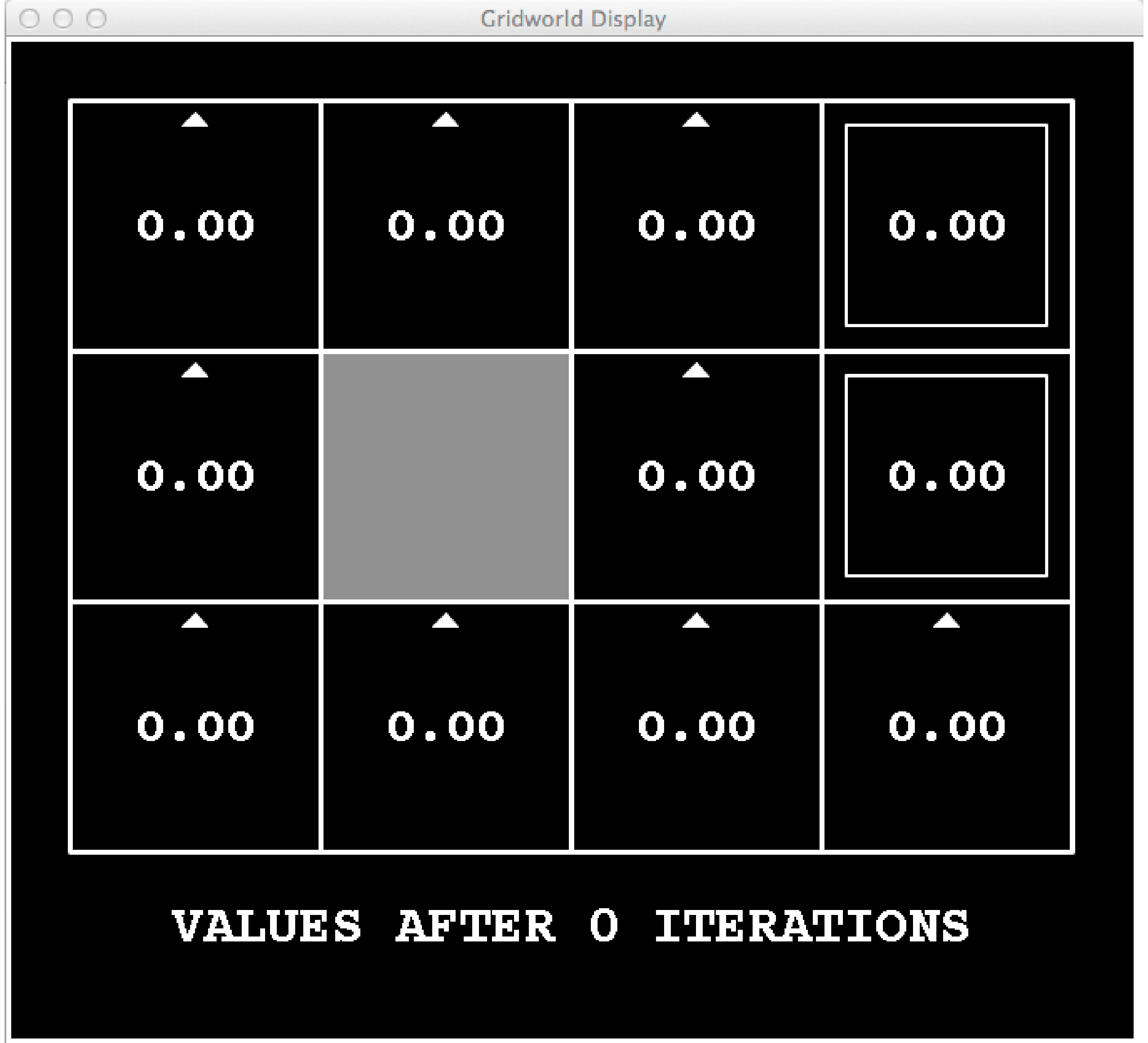
- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
  - Equivalently, it's what a depth-k expectimax would give from s





[Demo – time-limited values (L8D4)]

Gridworld: k=0

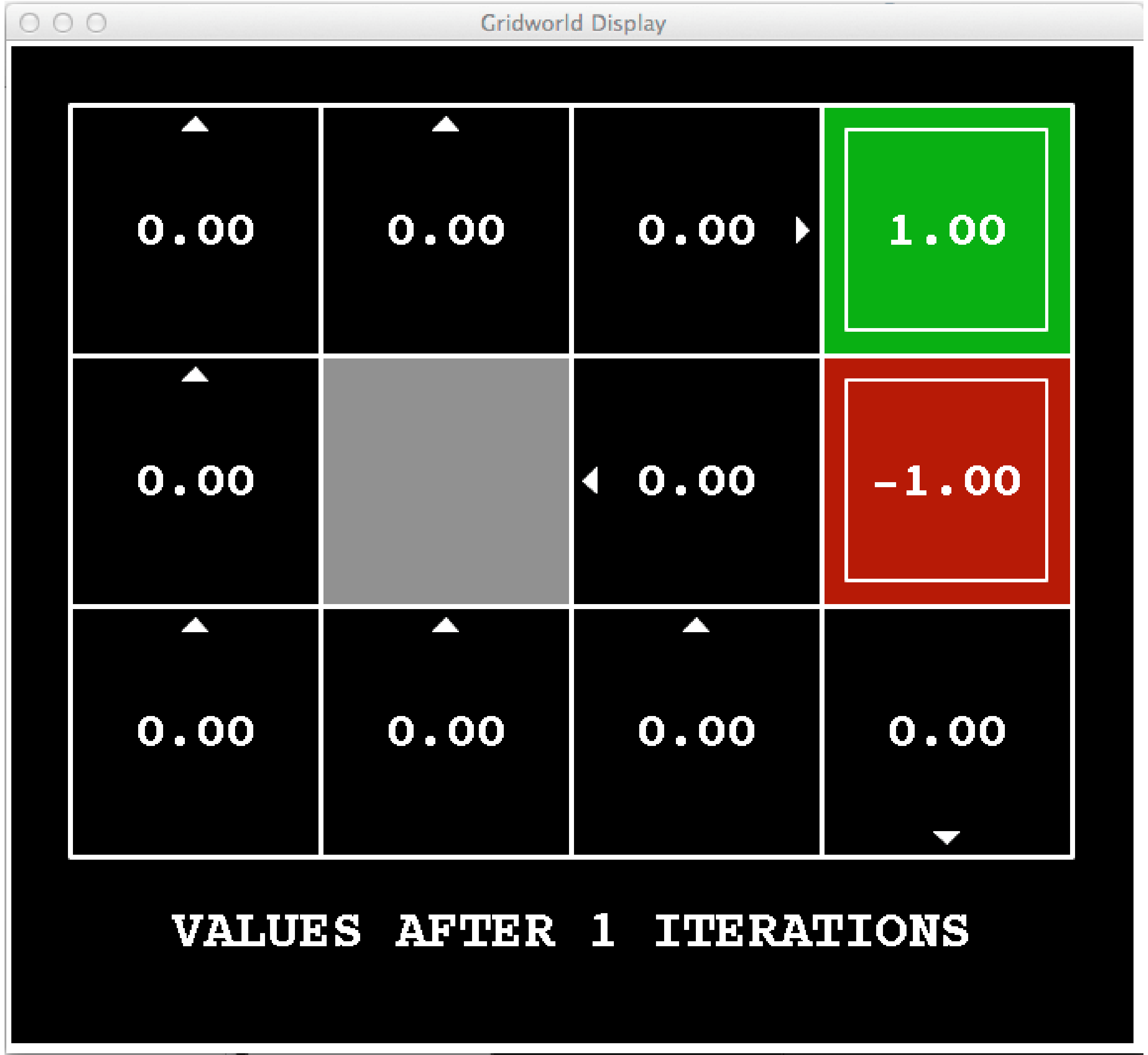


▲ 0.00	▲ 0.00	▲ 0.00	0.00
▲ 0.00		▲ 0.00	0.00
▲ 0.00	▲ 0.00	▲ 0.00	▲ 0.00

VALUES AFTER 0 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=1

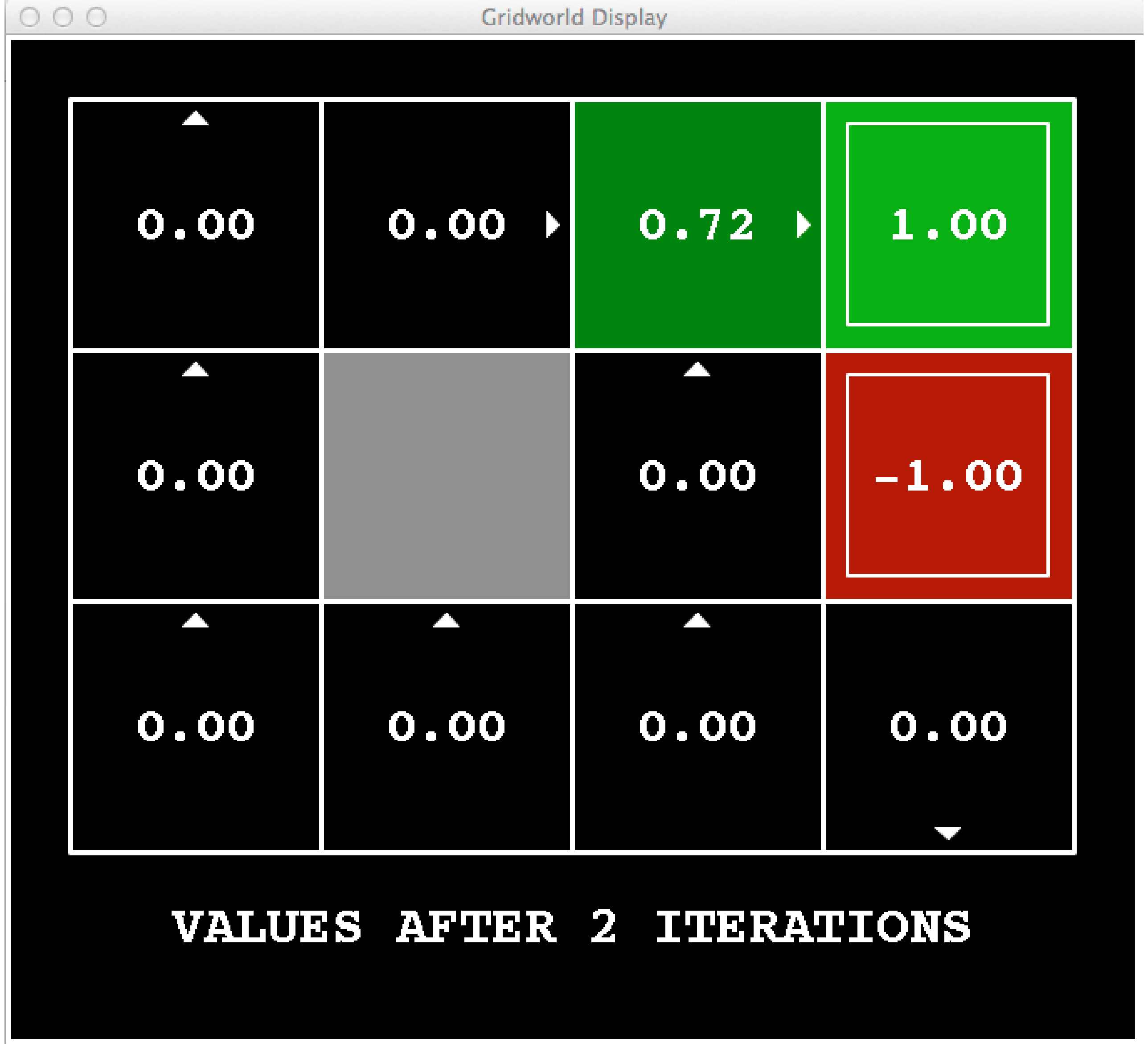


▲ 0.00	▲ 0.00	0.00 ▶	1.00
▲ 0.00		◀ 0.00	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 1 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=2

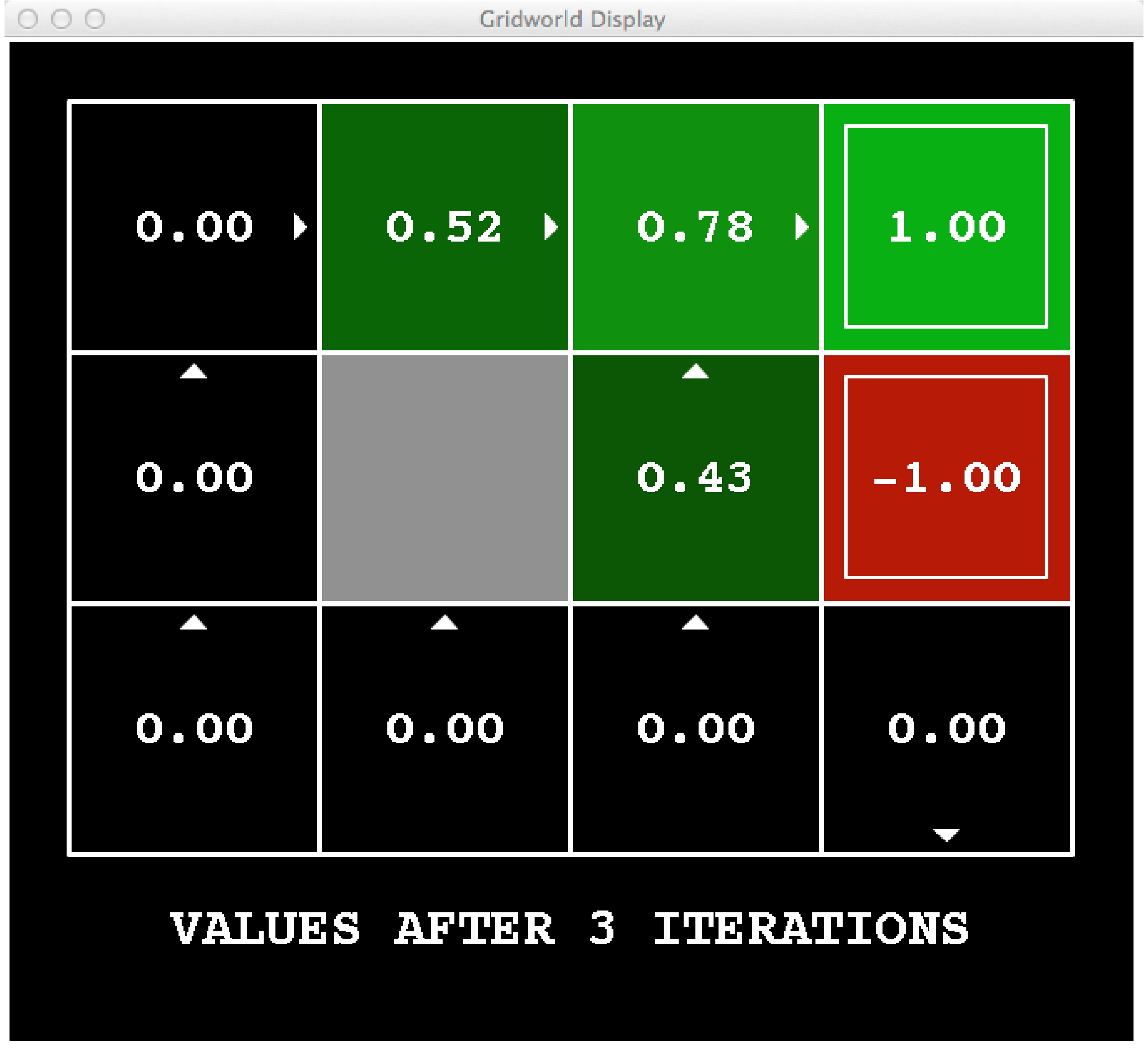


▲ 0.00	0.00 ▶	0.72 ▶	1.00
▲ 0.00		▲ 0.00	-1.00
▲ 0.00	▲ 0.00	▲ 0.00	0.00 ▼

VALUES AFTER 2 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=3

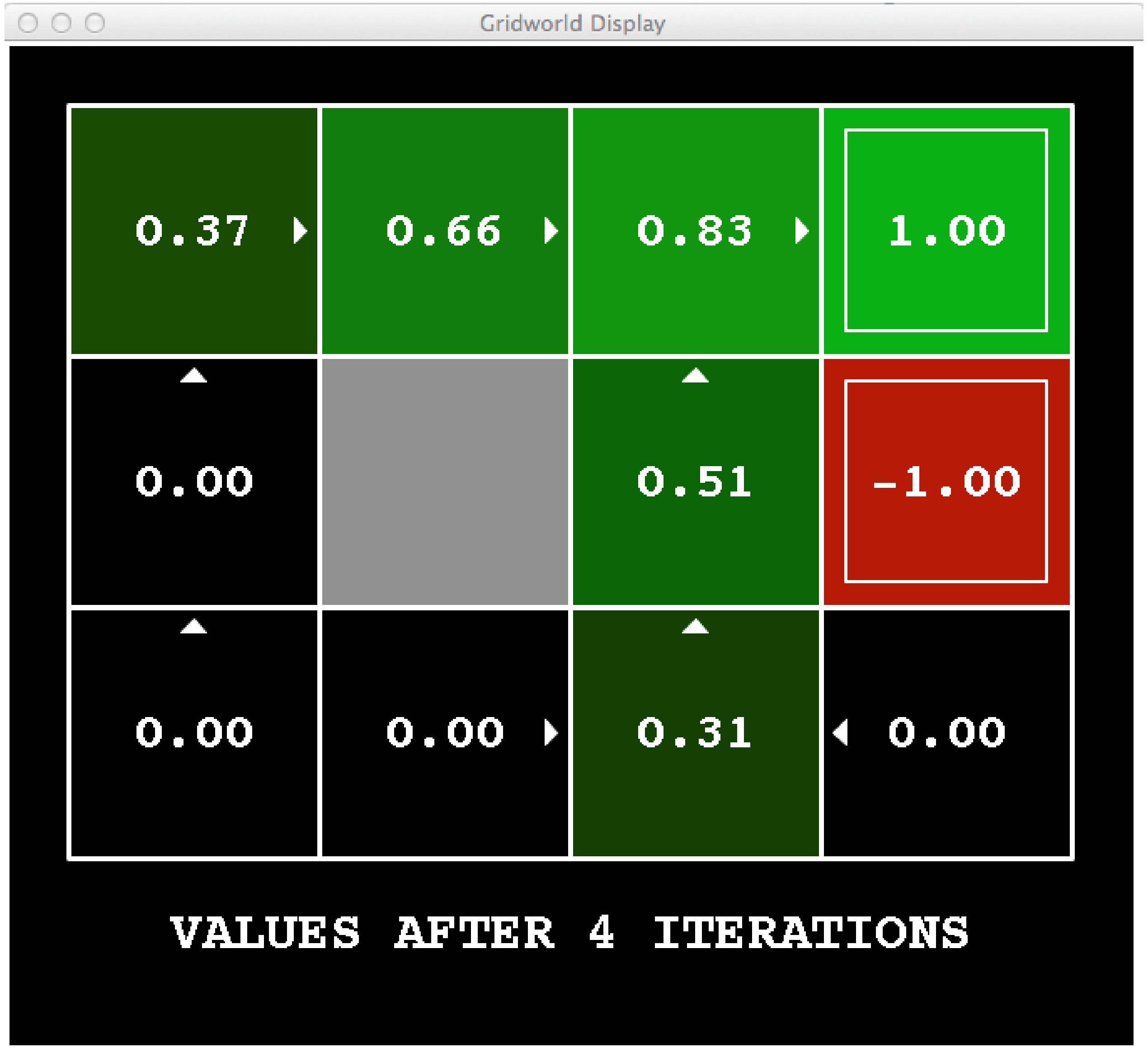


0.00 ▸	0.52 ▸	0.78 ▸	1.00
▴ 0.00		▴ 0.43	-1.00
▴ 0.00	▴ 0.00	▴ 0.00	0.00 ▾

VALUES AFTER 3 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=4



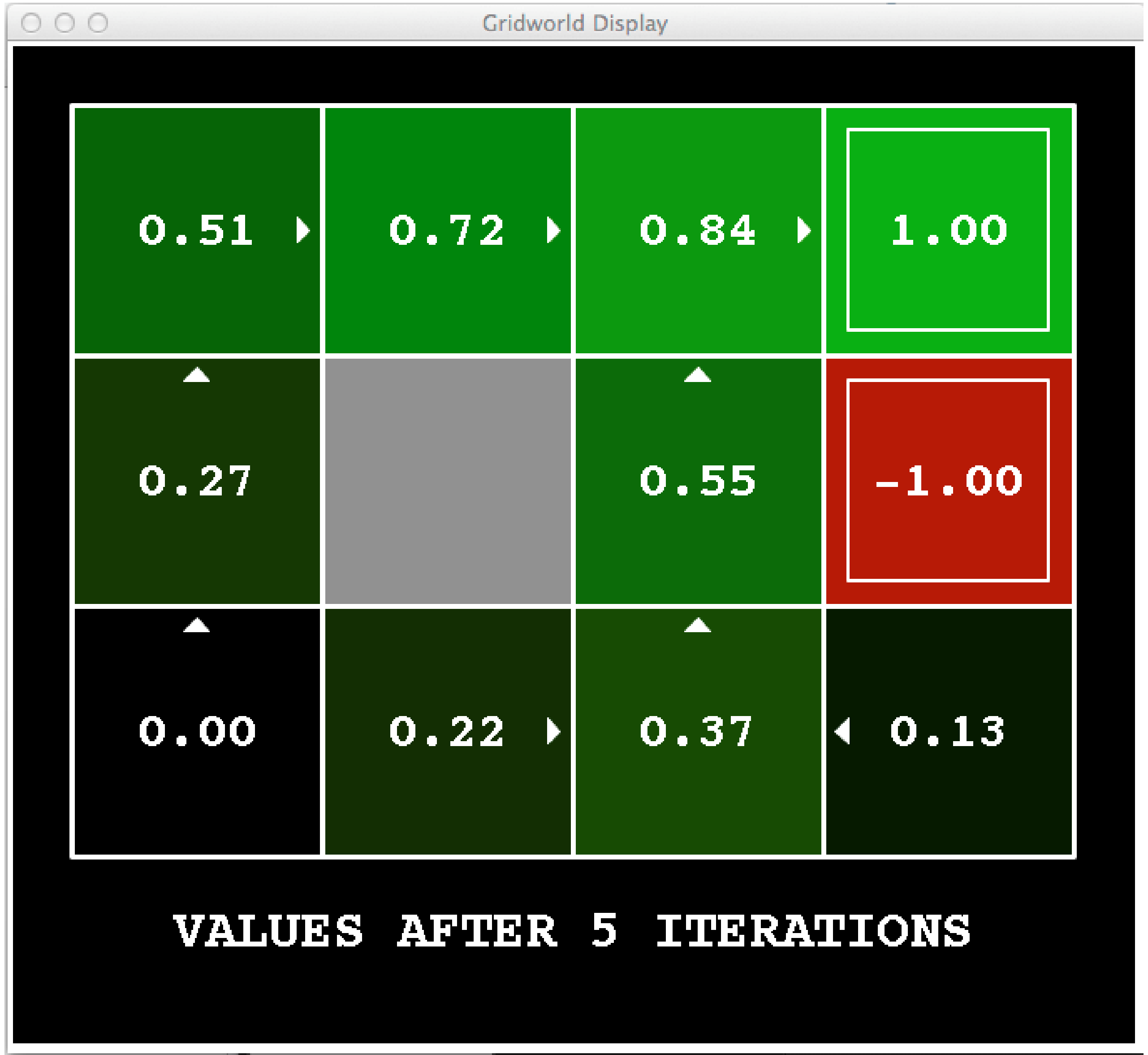
Gridworld Display

0.37 ▸	0.66 ▸	0.83 ▸	1.00
▴ 0.00		▴ 0.51	-1.00
▴ 0.00	0.00 ▸	▴ 0.31	◂ 0.00

VALUES AFTER 4 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=5

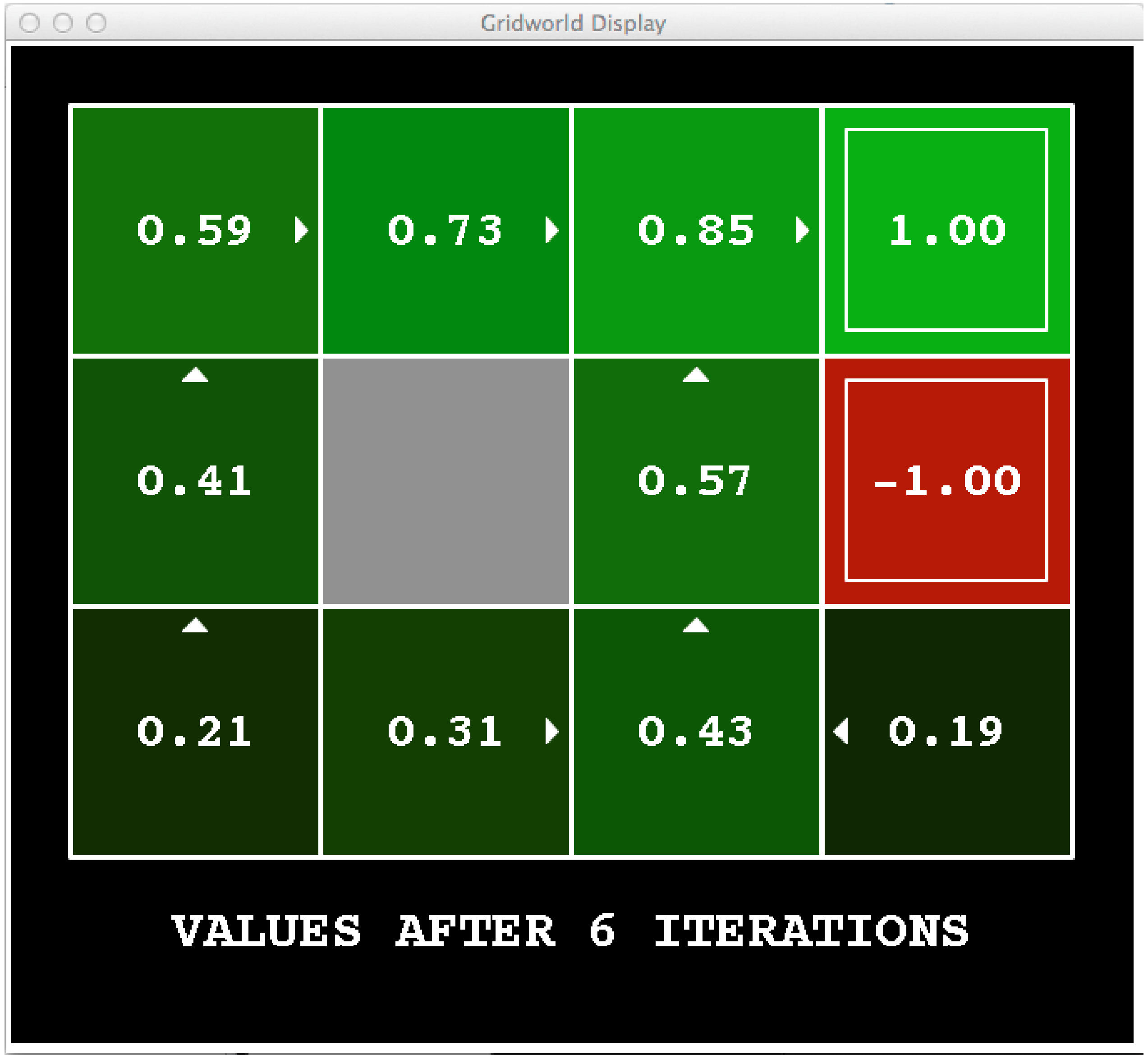


0.51 ▶	0.72 ▶	0.84 ▶	1.00
▲ 0.27		▲ 0.55	-1.00
▲ 0.00	0.22 ▶	▲ 0.37	◀ 0.13

VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=6

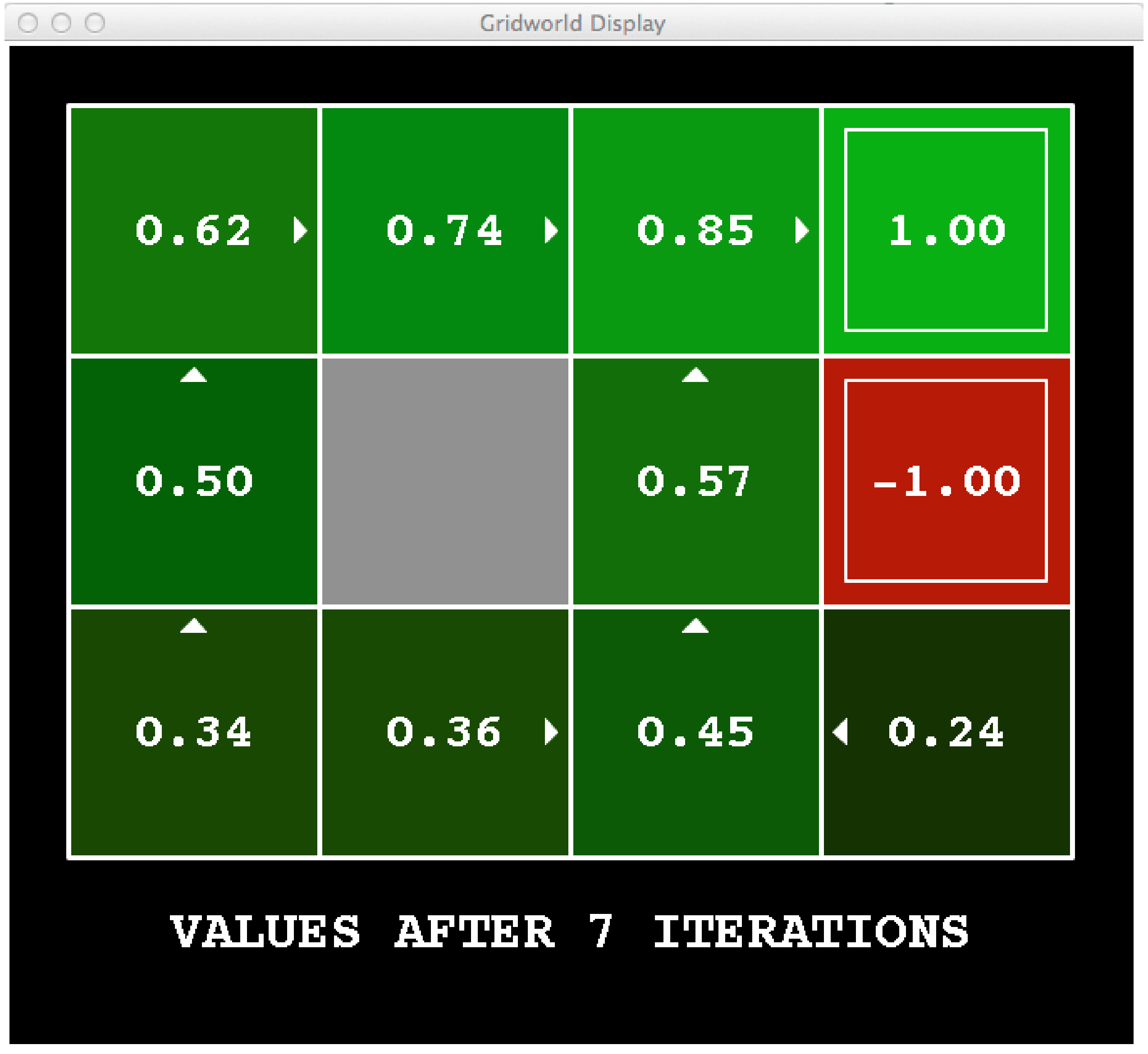


0.59 ▶	0.73 ▶	0.85 ▶	1.00
▲ 0.41		▲ 0.57	-1.00
▲ 0.21	0.31 ▶	▲ 0.43	◀ 0.19

VALUES AFTER 6 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=7

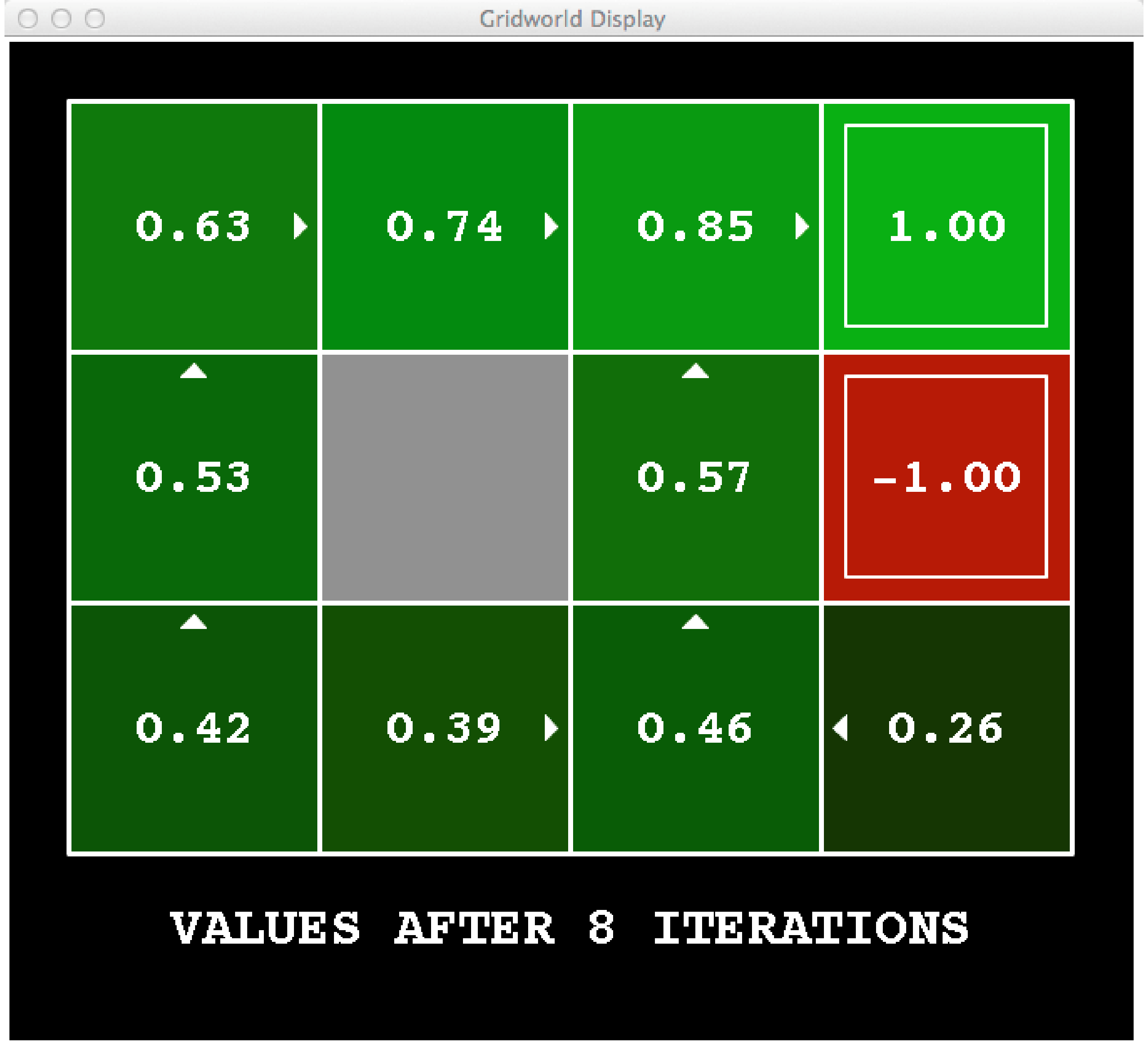


0.62 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.50		▴ 0.57	-1.00
▴ 0.34	0.36 ▸	▴ 0.45	◂ 0.24

VALUES AFTER 7 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=8

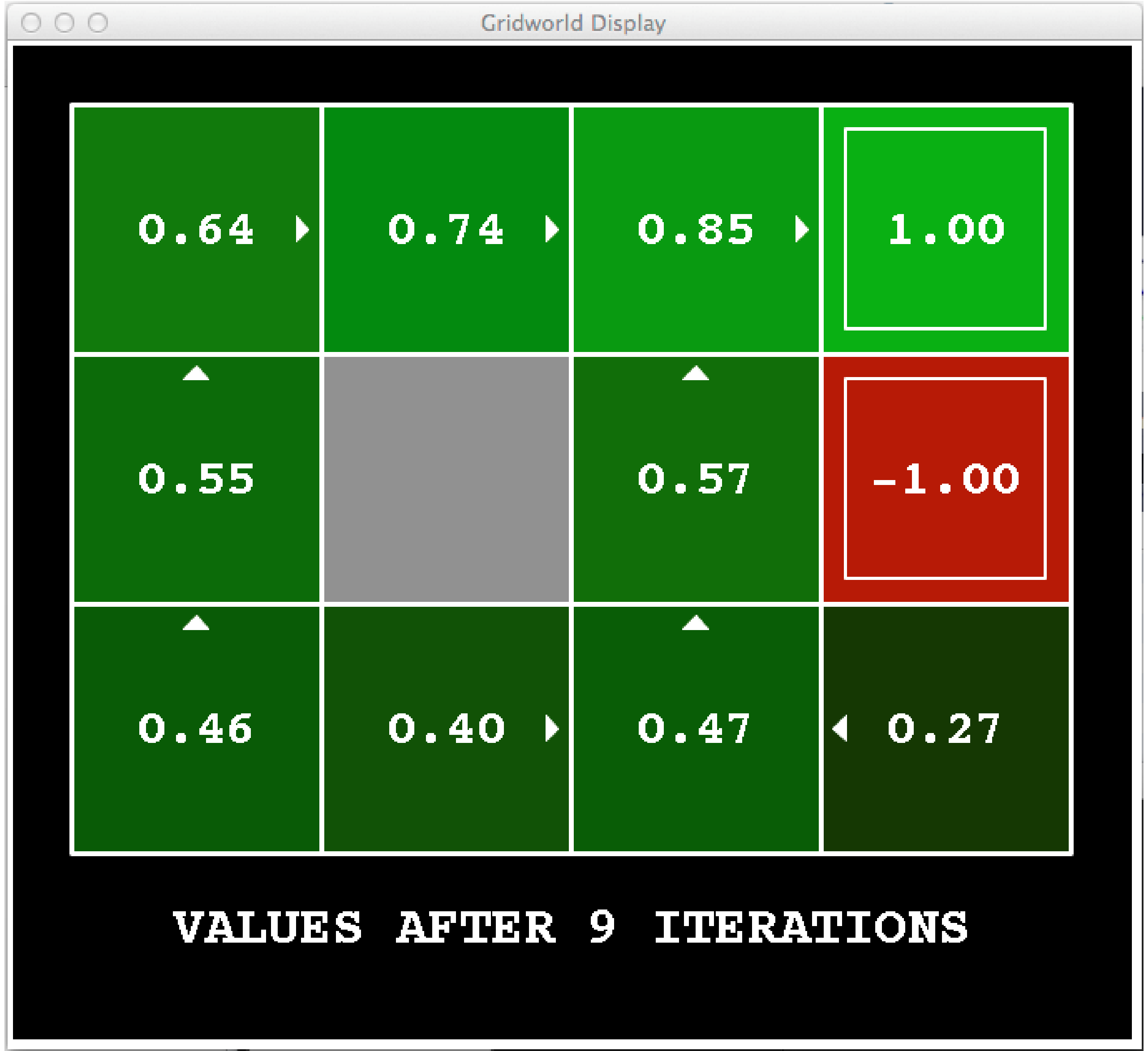


0.63 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.53		▴ 0.57	-1.00
▴ 0.42	0.39 ▸	▴ 0.46	◂ 0.26

VALUES AFTER 8 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=9

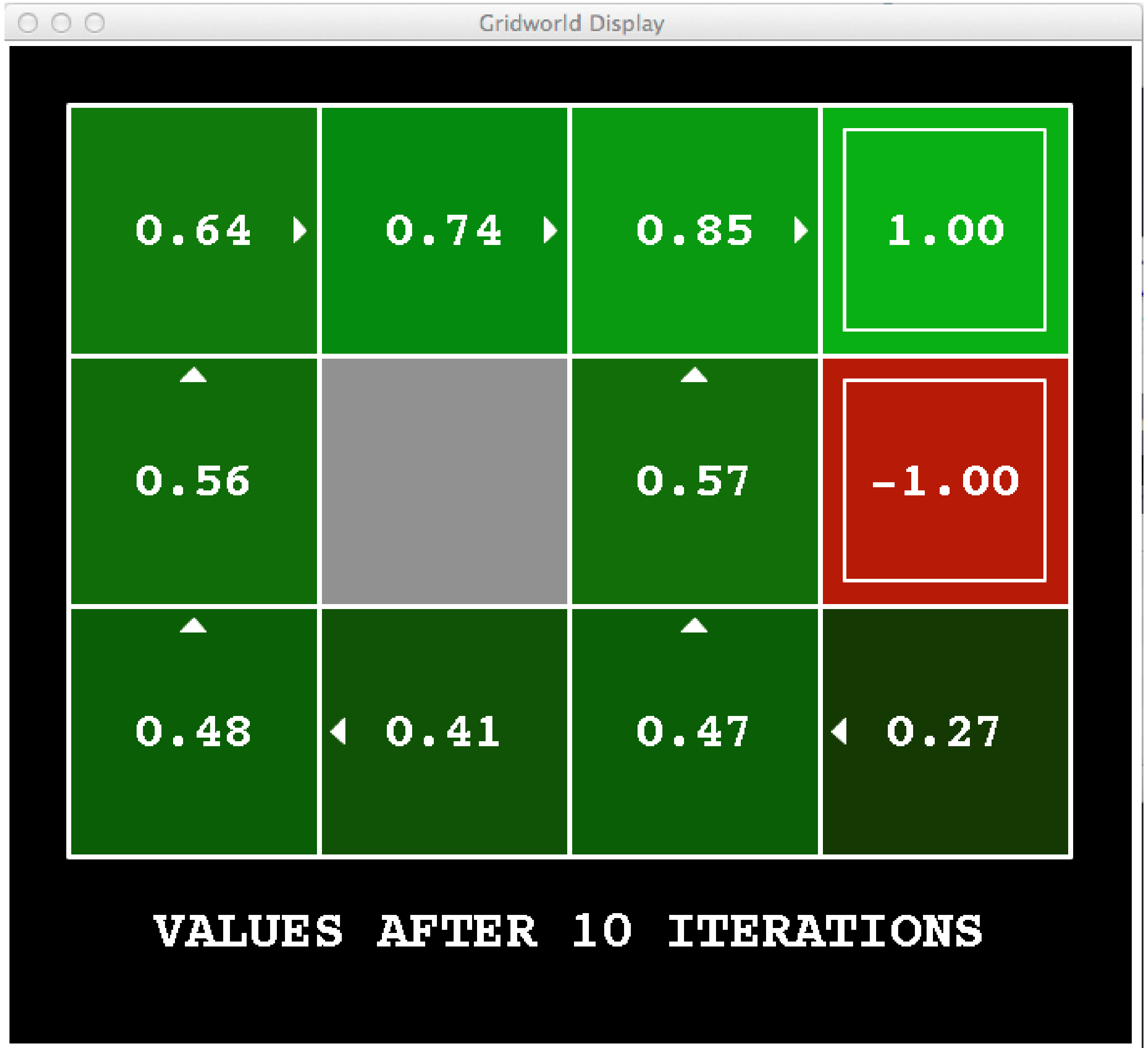


0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.55		▲ 0.57	-1.00
▲ 0.46	0.40 ▶	▲ 0.47	◀ 0.27

VALUES AFTER 9 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=10

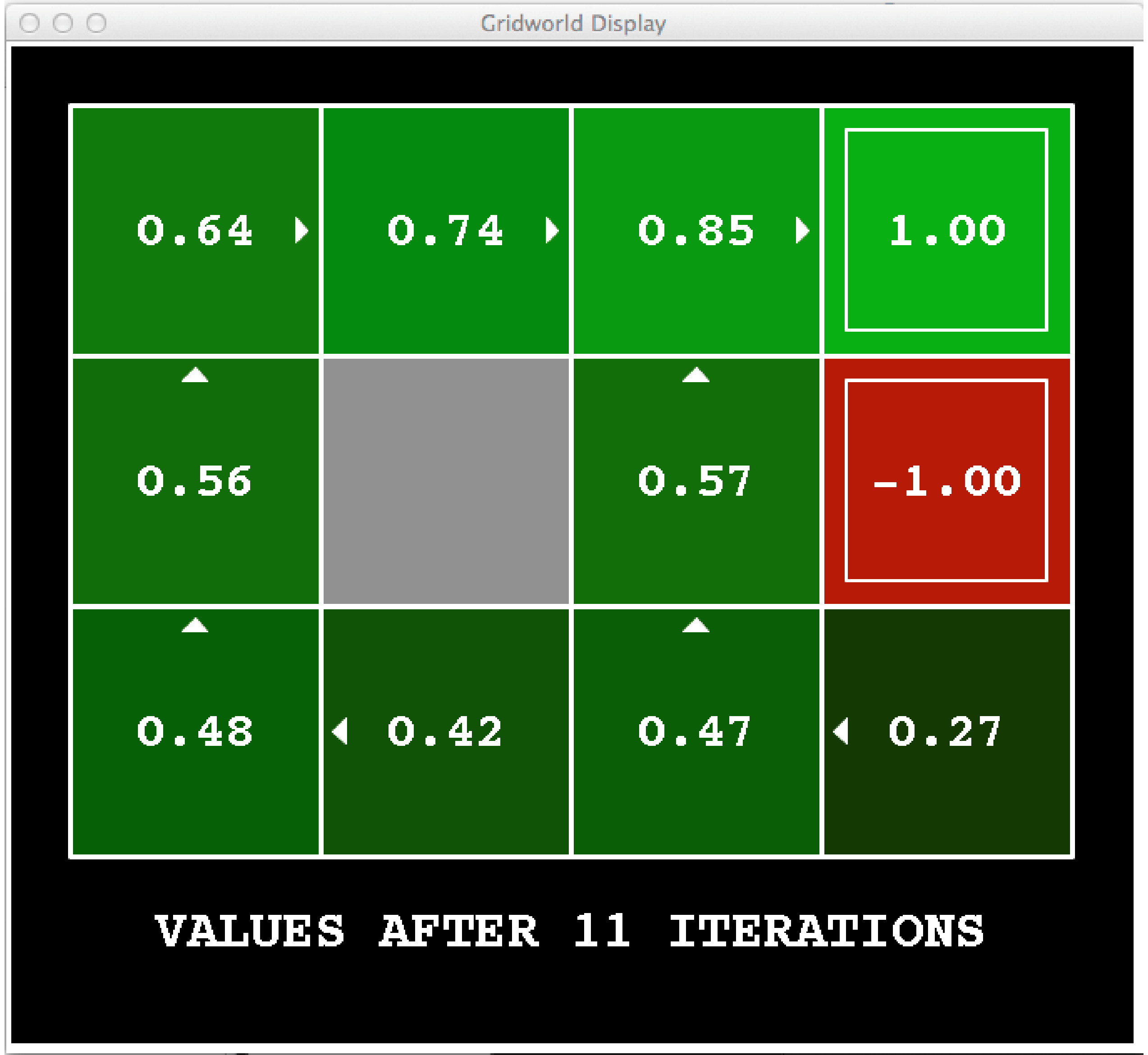


0.64 ▶	0.74 ▶	0.85 ▶	1.00
▲ 0.56		▲ 0.57	-1.00
▲ 0.48	◀ 0.41	▲ 0.47	◀ 0.27

VALUES AFTER 10 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=11

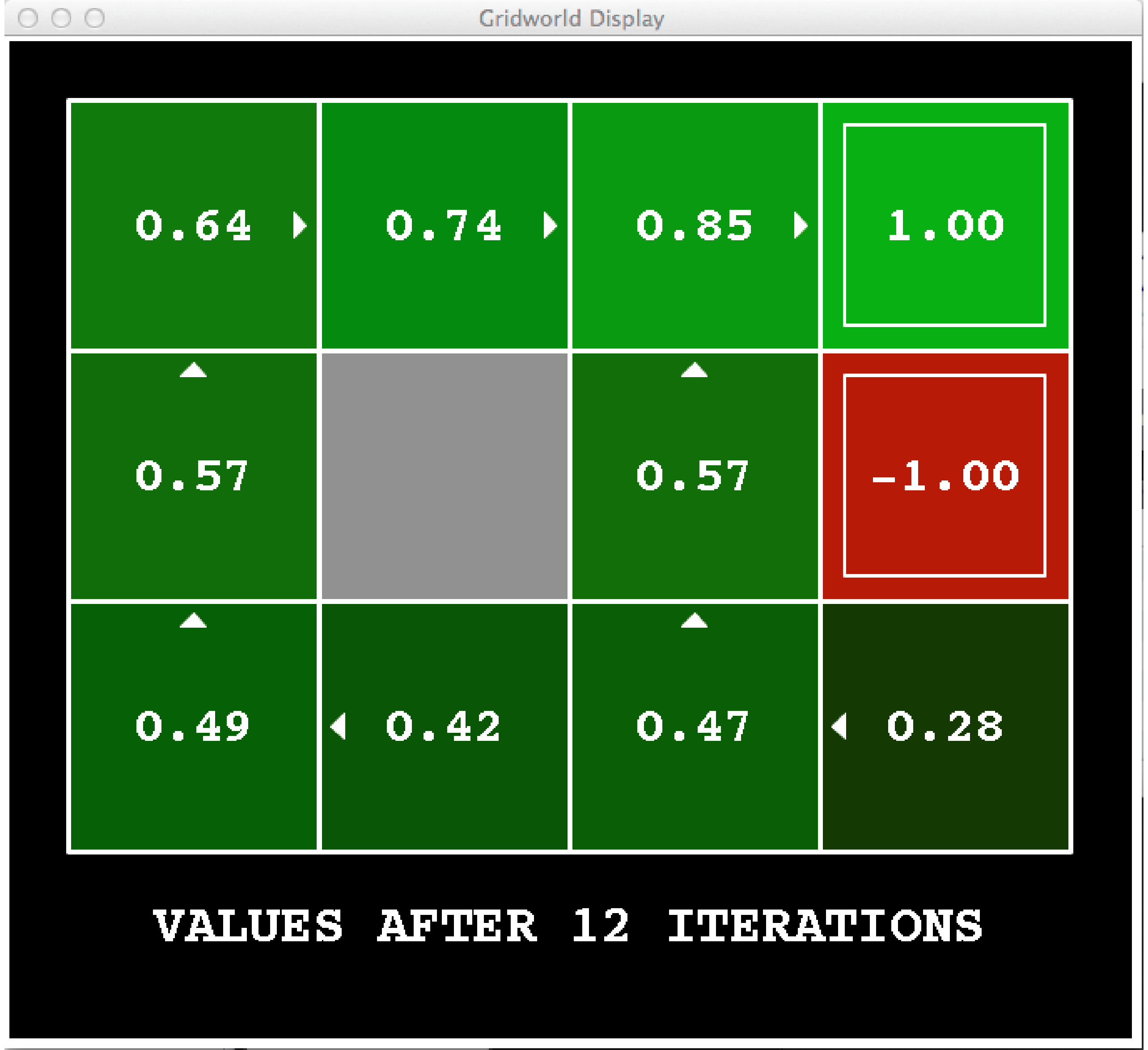


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▲ 0.56		▲ 0.57	-1.00
▲ 0.48	◂ 0.42	▲ 0.47	◂ 0.27

VALUES AFTER 11 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=12

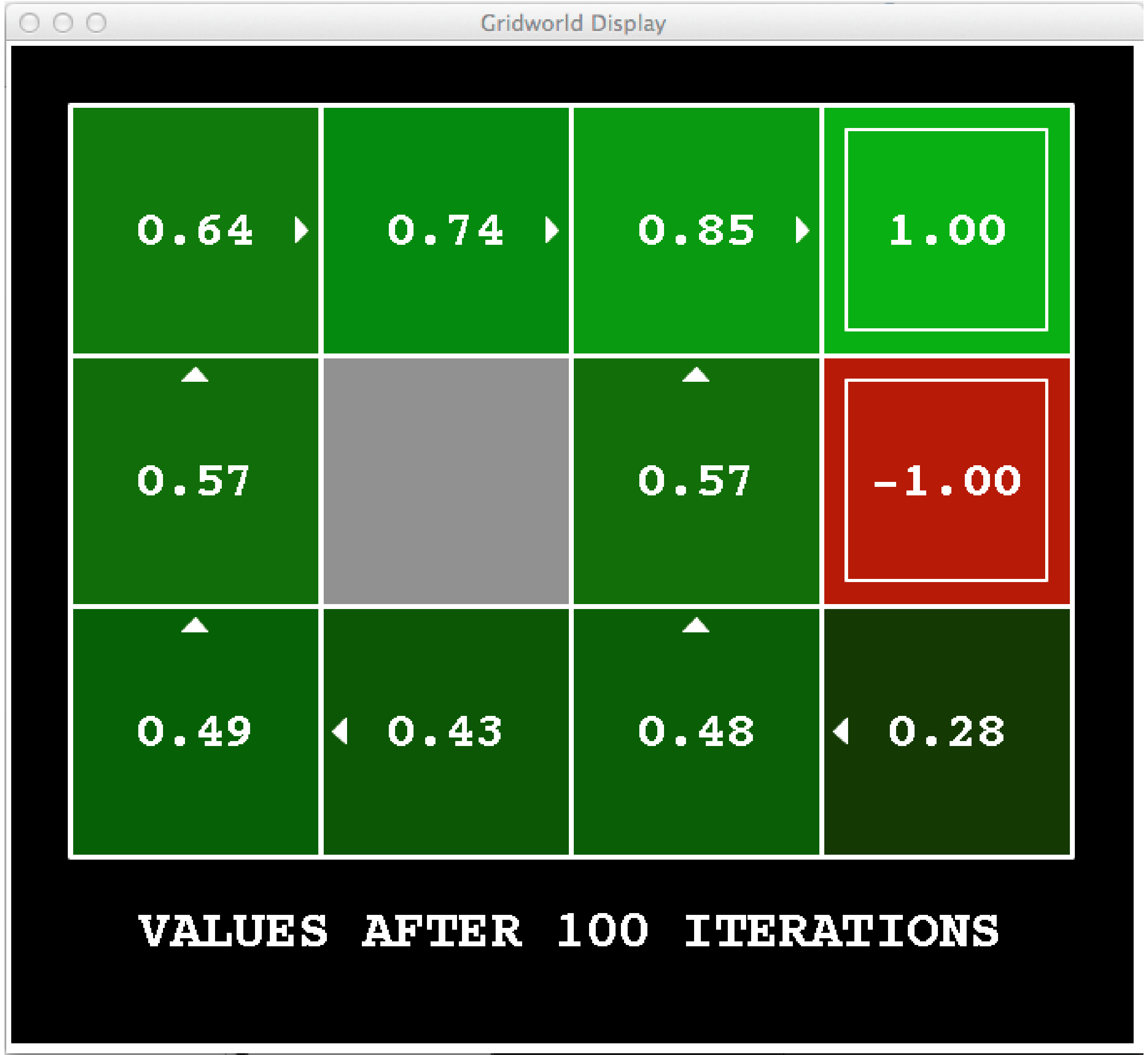


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.57		▴ 0.57	-1.00
▴ 0.49	◂ 0.42	▴ 0.47	◂ 0.28

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=100



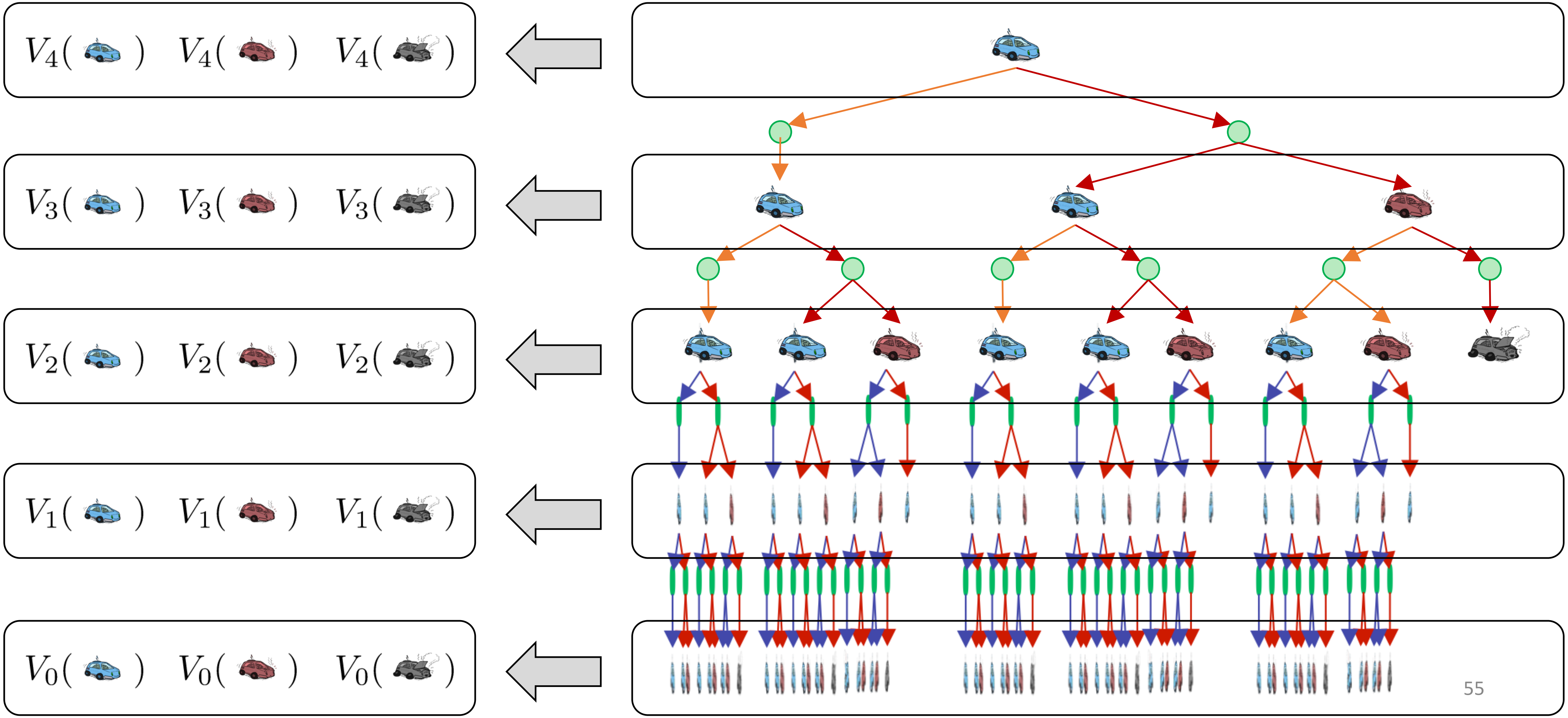
Gridworld Display

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

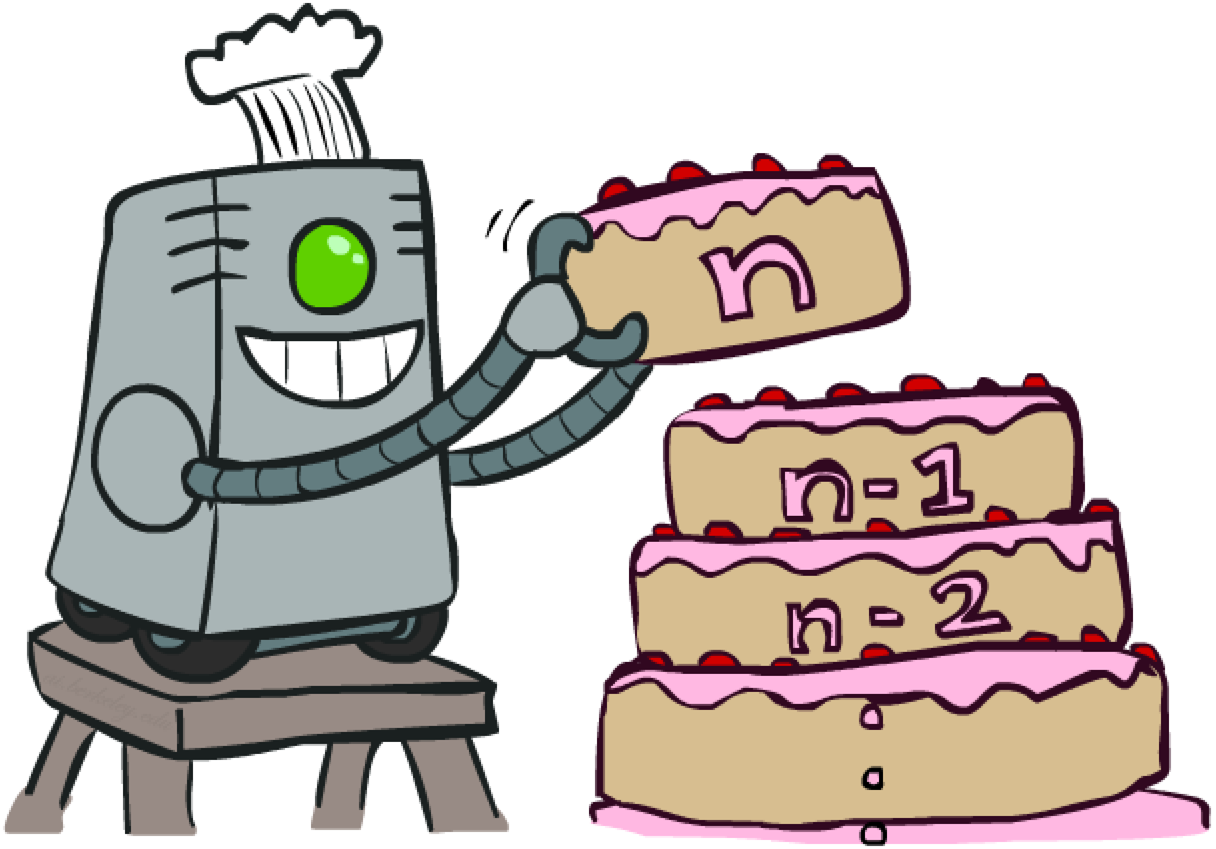
VALUES AFTER 100 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Time-Limited Values: Computing


$V_4(\quad)$ $V_4(\quad)$ $V_4(\quad)$
$V_3(\quad)$ $V_3(\quad)$ $V_3(\quad)$
$V_2(\quad)$ $V_2(\quad)$ $V_2(\quad)$
$V_1(\quad)$ $V_1(\quad)$ $V_1(\quad)$
$V_0(\quad)$ $V_0(\quad)$ $V_0(\quad)$

Value Iteration

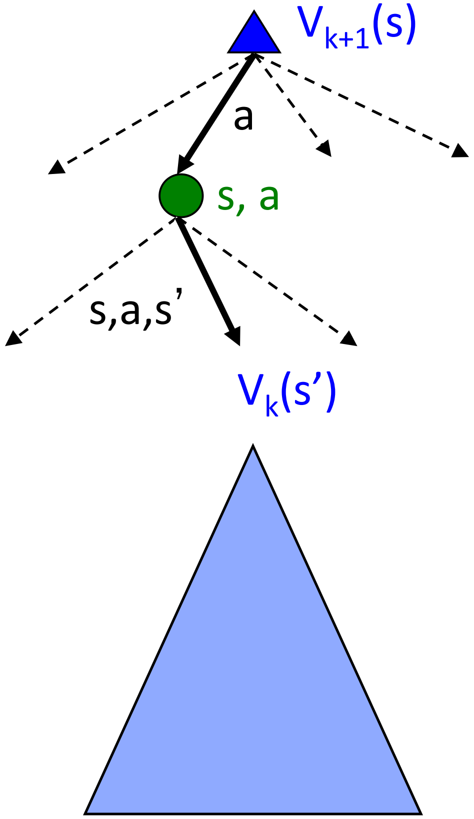


Value Iteration

- 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, which yields V*
- 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



Example

V_2

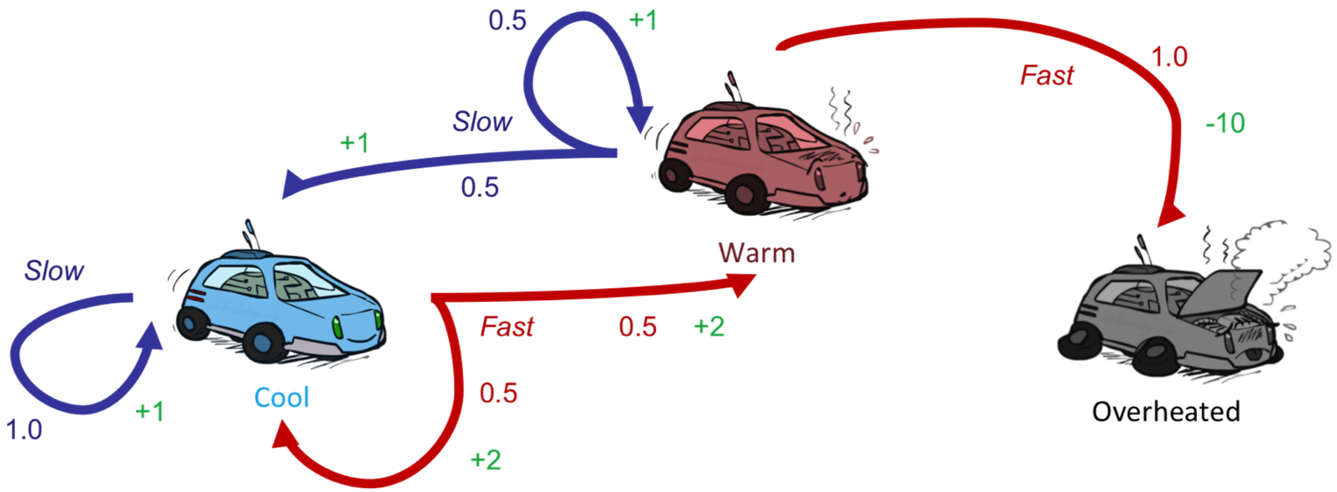
V_1

S: 1

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

V_0

0 0 0



Assume no discount!

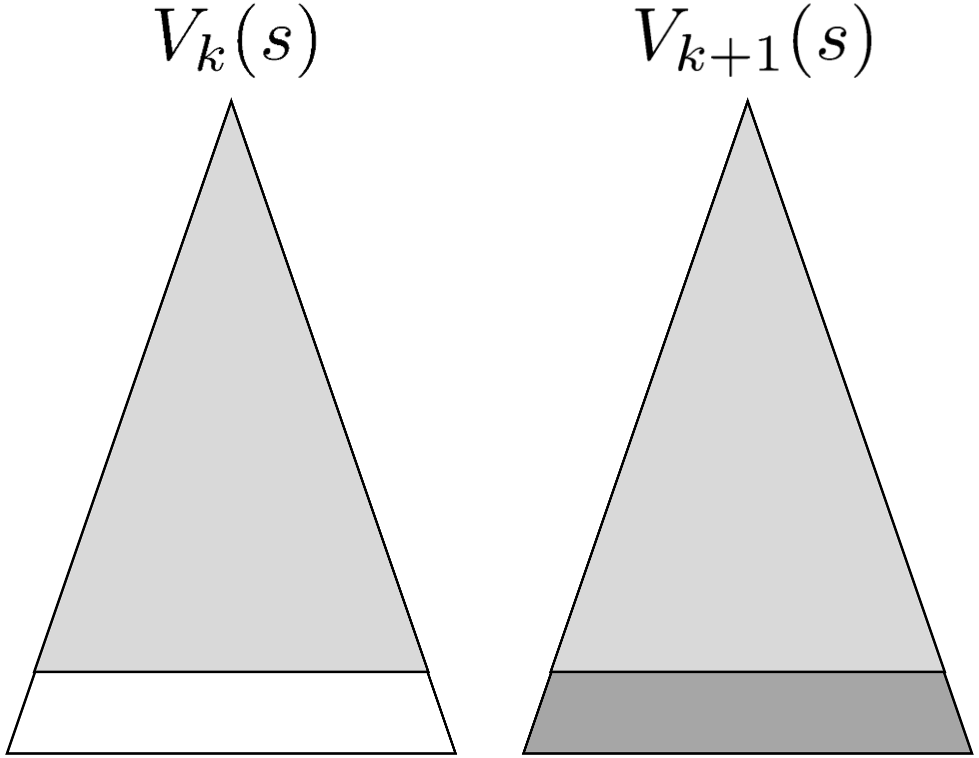
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

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

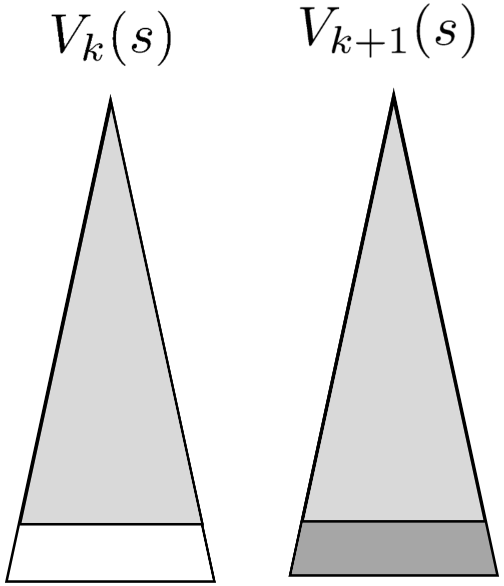
- Case 2: If the discount is less than 1
- Proof Sketch:
  - For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
  - That last layer is at best all R_{MAX}
  - 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



Convergence 2

- $V_1(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_0(s')]$
  $V_1'(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_0'(s')]$
- If $|V_0(s) - V_0'(s)| \leq \epsilon$, then
  $$|V_1(s) - V_1'(s)| \leq \gamma\epsilon$$
- Note $\left|V_1(s) = \max_a \sum_{s'} T(s,a,s')R(s,a,s')\right| \leq R_{\max}$

  that is $|V_1(s) - V_0(s)| \leq R_{\max}$, then $|V_2(s) - V_1(s)| \leq \gamma R_{\max}$ and
  $$|V_{k+1}(s) - V_k(s)| \leq \gamma^k R_{\max}$$



Value Iteration (Revisited)

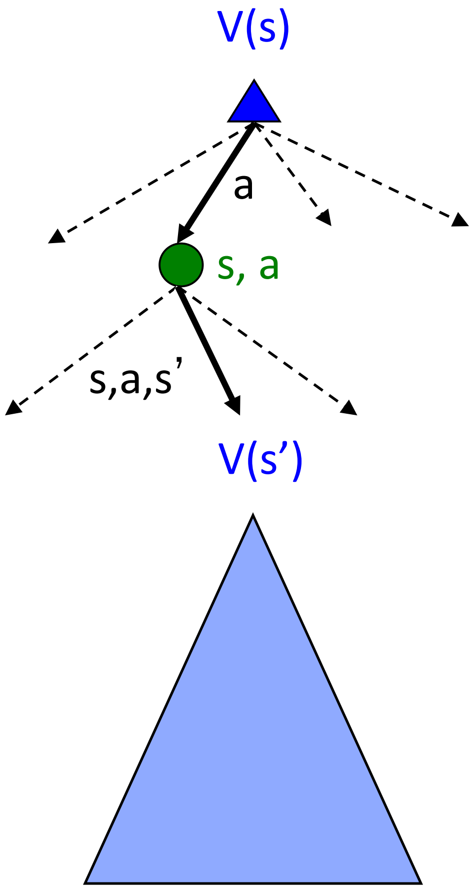
- Bellman equations characterize the optimal values:

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

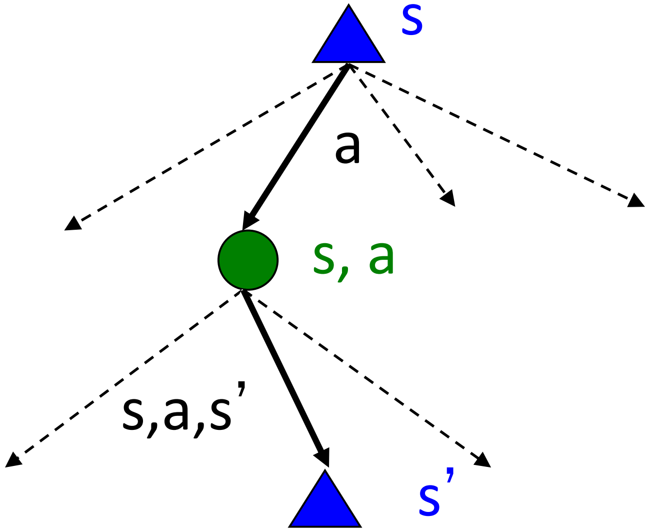
- Value iteration computes them:

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

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



Value Iteration - Implementation



- Init:
  - $\forall s: \quad V(s) = 0$
- Iterate:
  - $\forall s: \quad V_{new}(s) = \max_{a} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
  - $V = V_{new}$

Note: can even directly assign to *V(s)*, which will not compute the sequence of V_k but will still converge to *V**

同步 vs. 异步价值迭代

□ 同步的价值迭代会储存两份价值函数的拷贝

1. 对S中的所有状态s

$$V_{new}(s) \leftarrow \max_{a\in A} \sum_{s'\in S} P_{s,a}(s')[R(s,a,s') + \gamma V_{old}(s')]$$

2. 更新 $V_{old}(s) \leftarrow V_{new}(s)$

□ 异步价值迭代只储存一份价值函数

1. 对S中的所有状态s

$$V(s) \leftarrow \max_{a\in A} \sum_{s'\in S} P_{s,a}(s')[R(s,a,s') + \gamma V(s')]$$

价值迭代例子：最短路径

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

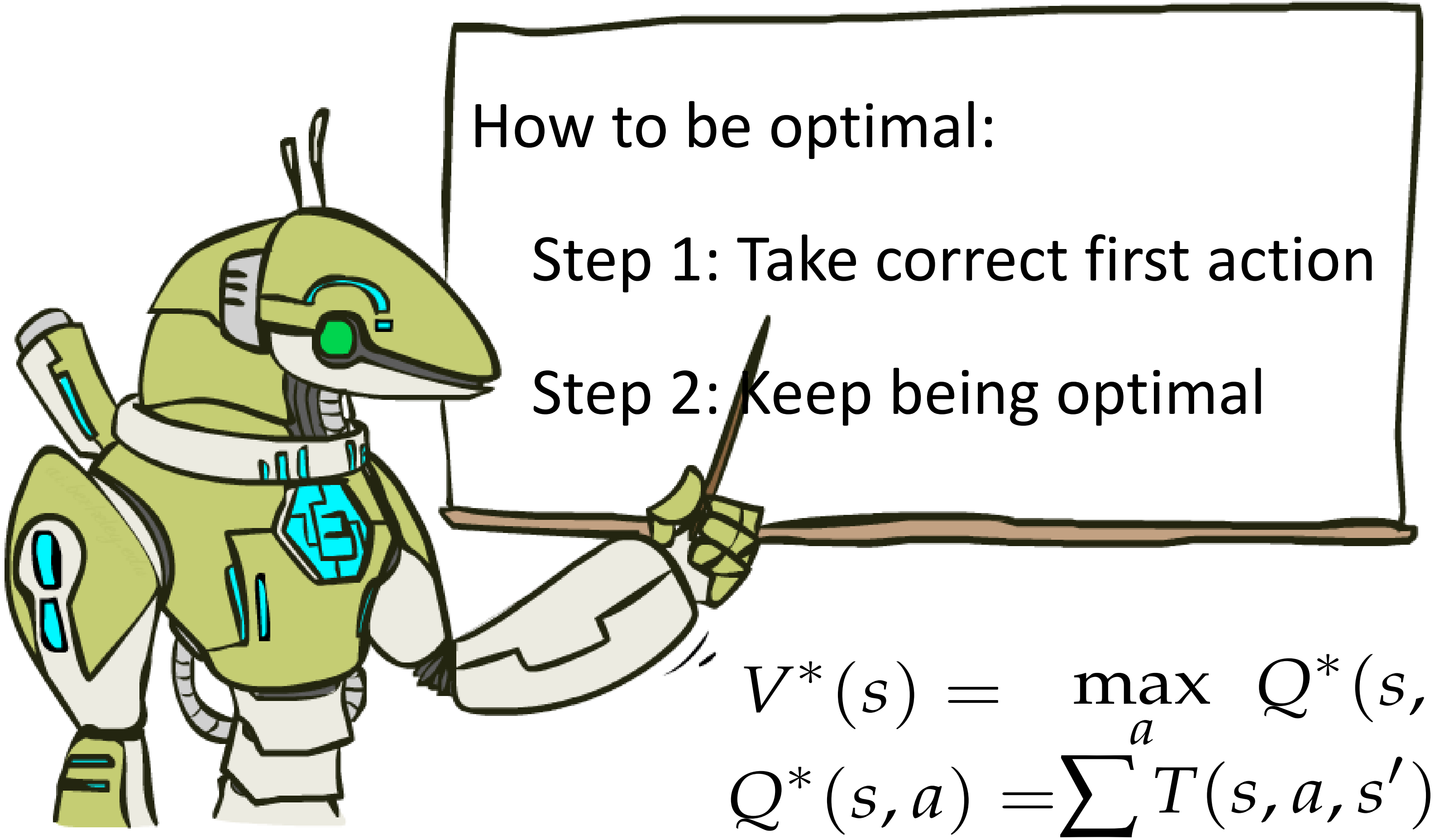
0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

The Bellman Equations



How to be optimal:

Step 1: Take correct first action

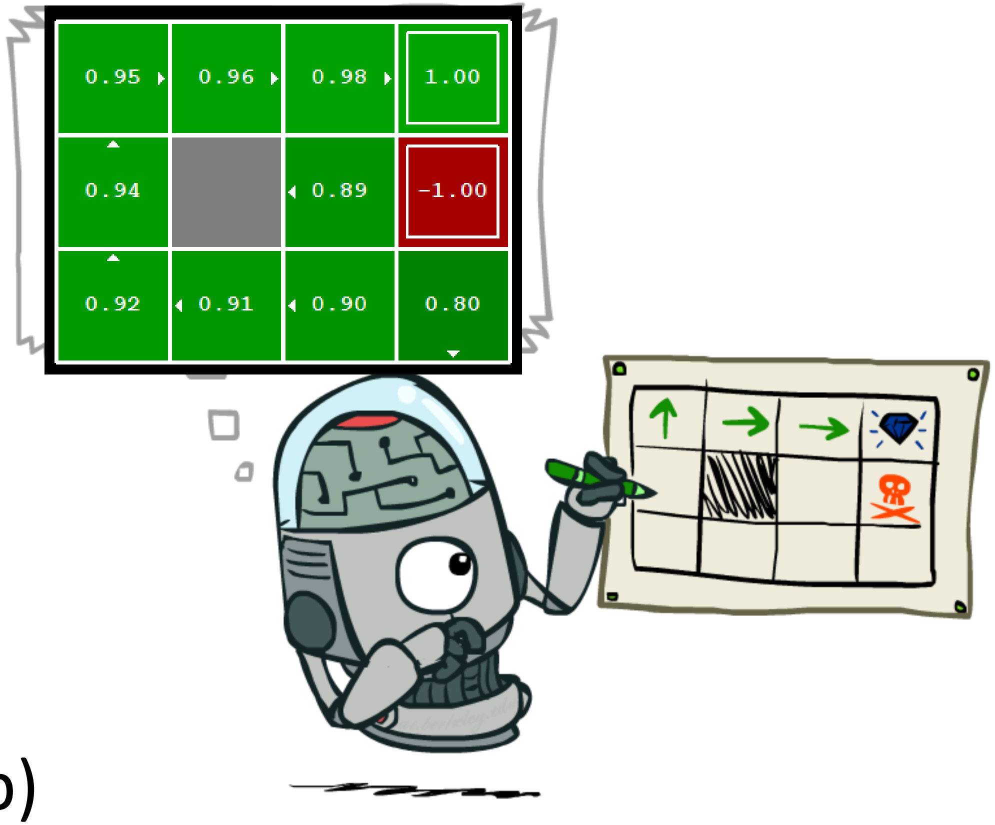
Step 2: Keep being optimal

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

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

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

Policy Extraction: Computing Actions from Values



0.95 ▸	0.96 ▸	0.98 ▸	1.00
▴ 0.94	▨	◂ 0.89	-1.00
▴ 0.92	◂ 0.91	◂ 0.90	0.80 ▾

- Let’s imagine we have the optimal values V*(s)

- How should we act?
  - It’s not obvious!

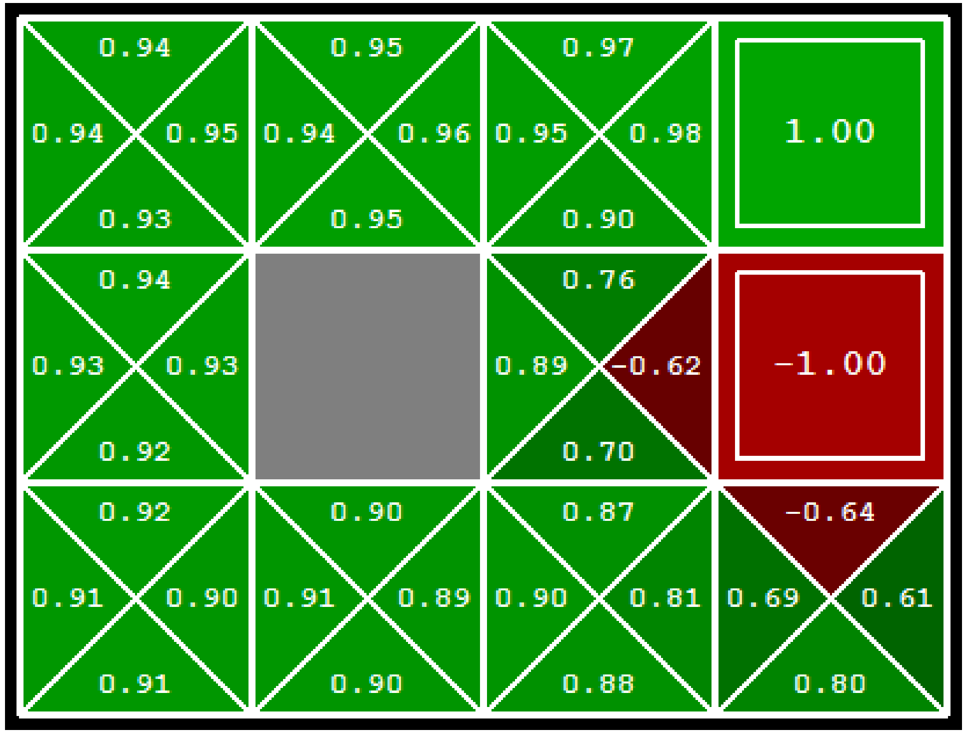
- We need to do a mini-expectimax (one step)

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

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

Policy Extraction: Computing Actions from Q-Values

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

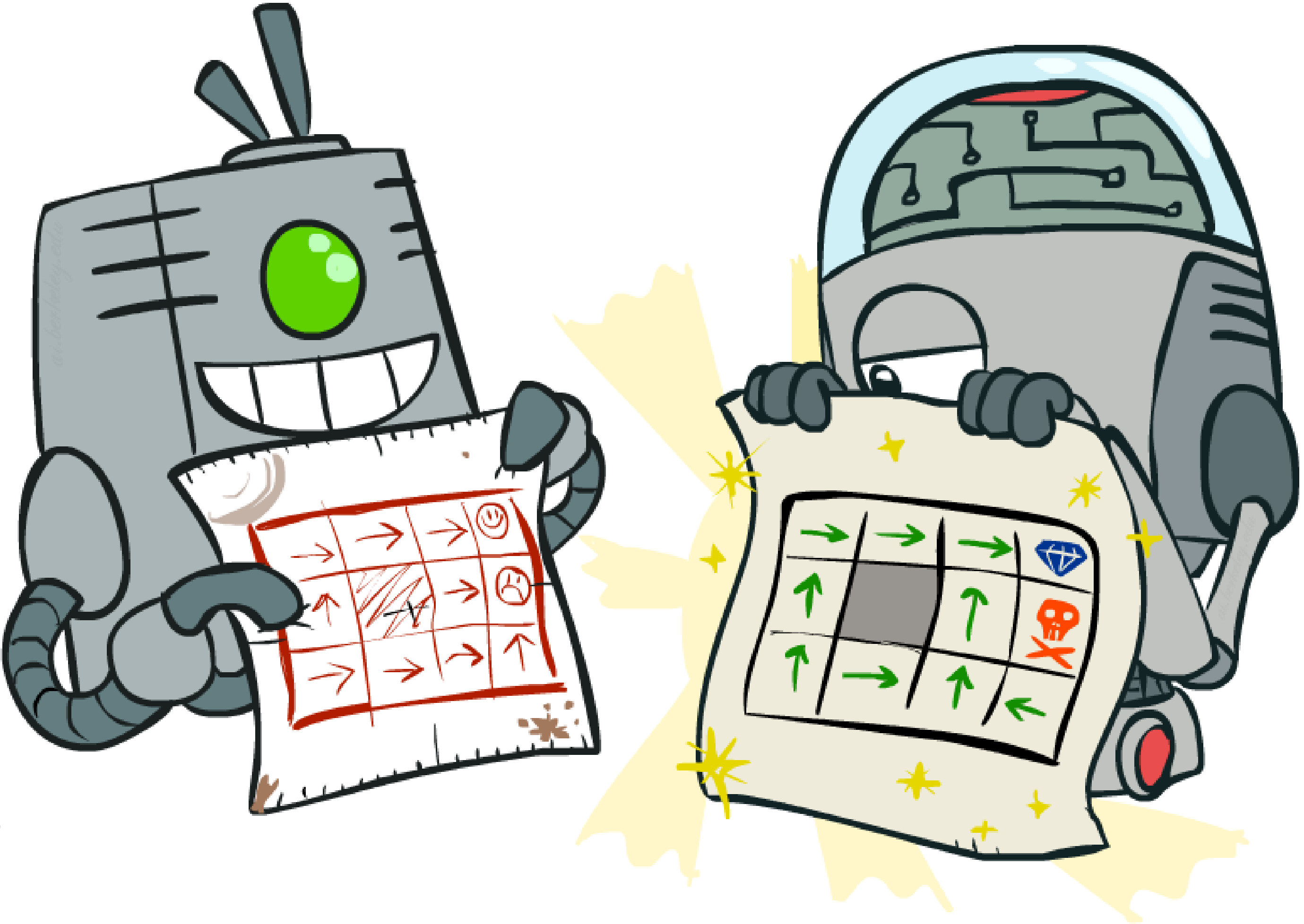


- How should we act?
  - Completely trivial to decide!

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

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

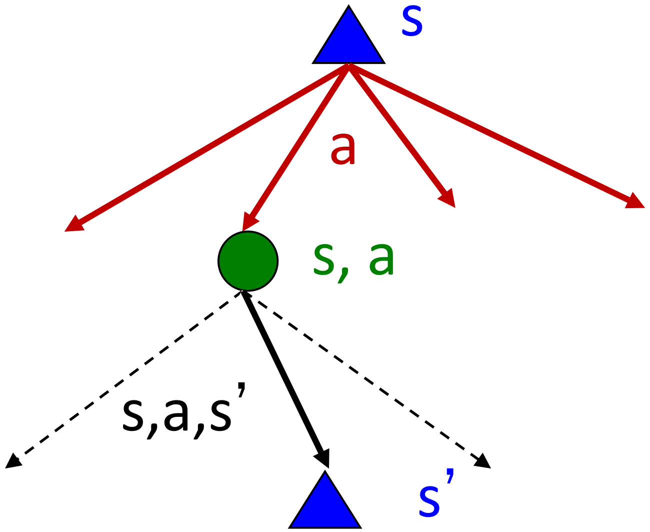
Policy Methods



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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

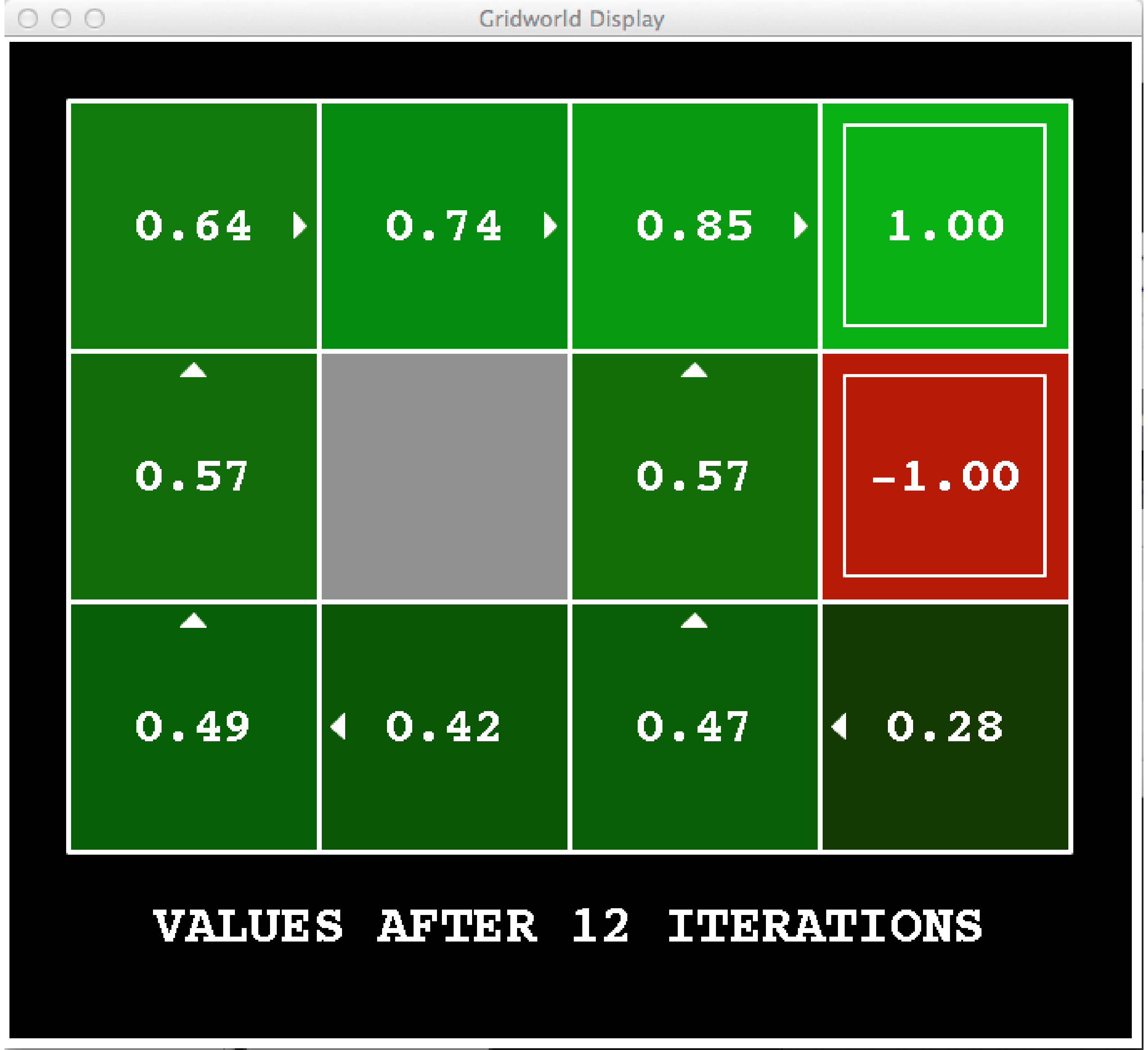


- Problem 1: It's slow – O(S^2A) per iteration

- Problem 2: The "max" at each state rarely changes

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

Gridworld: k=12

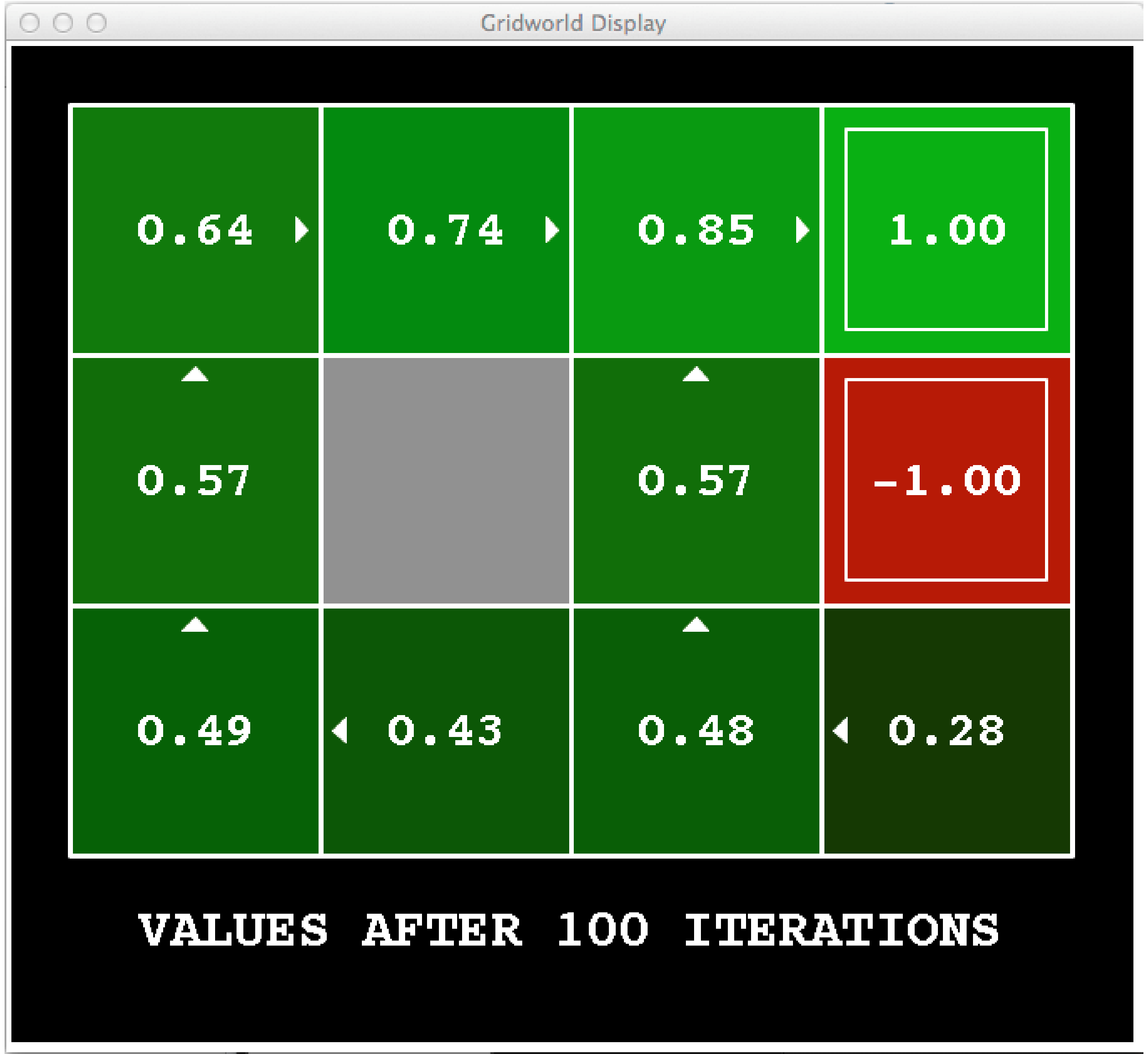


0.64 ▸	0.74 ▸	0.85 ▸	1.00
▴ 0.57		▴ 0.57	-1.00
▴ 0.49	◂ 0.42	▴ 0.47	◂ 0.28

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld: k=100



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

VALUES AFTER 100 ITERATIONS

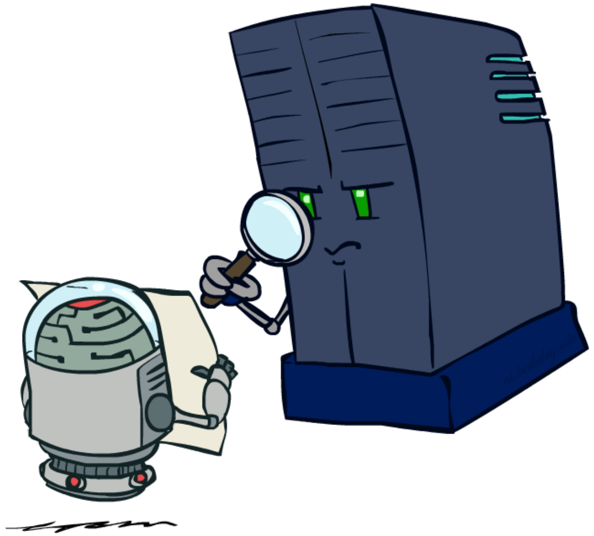
Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

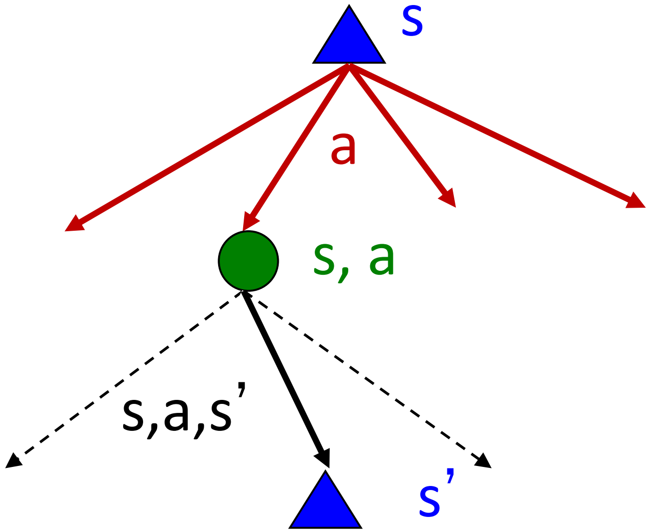
- Alternative approach for optimal values:
  - Step 1: Policy Evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy Improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values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

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

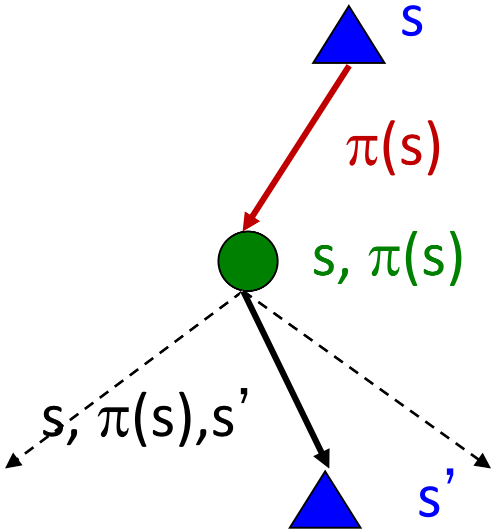
Policy Evaluation: Fixed Policies



Do the optimal action



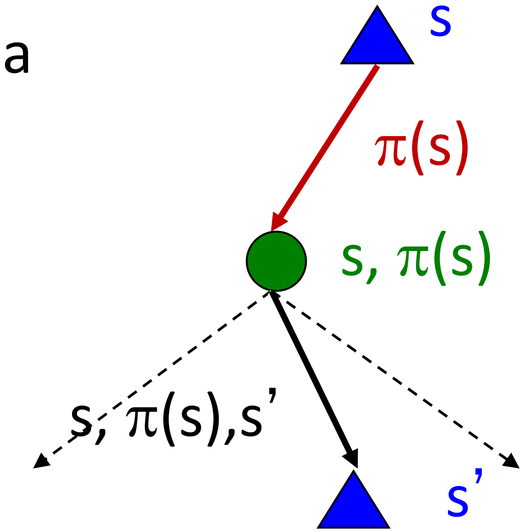
Do what π says to do



- Expectimax trees max over all actions to compute the optimal values
- If we fix 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

Policy Evaluation: Utilities for a Fixed Policy

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



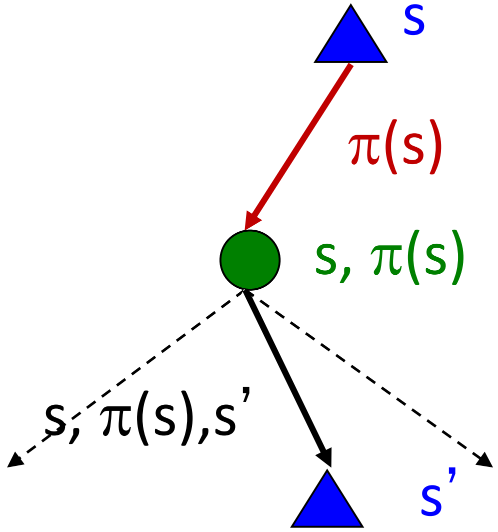
- Define the utility of a state s, under a fixed policy π:

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

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

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

Policy Evaluation: Implementation



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

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

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

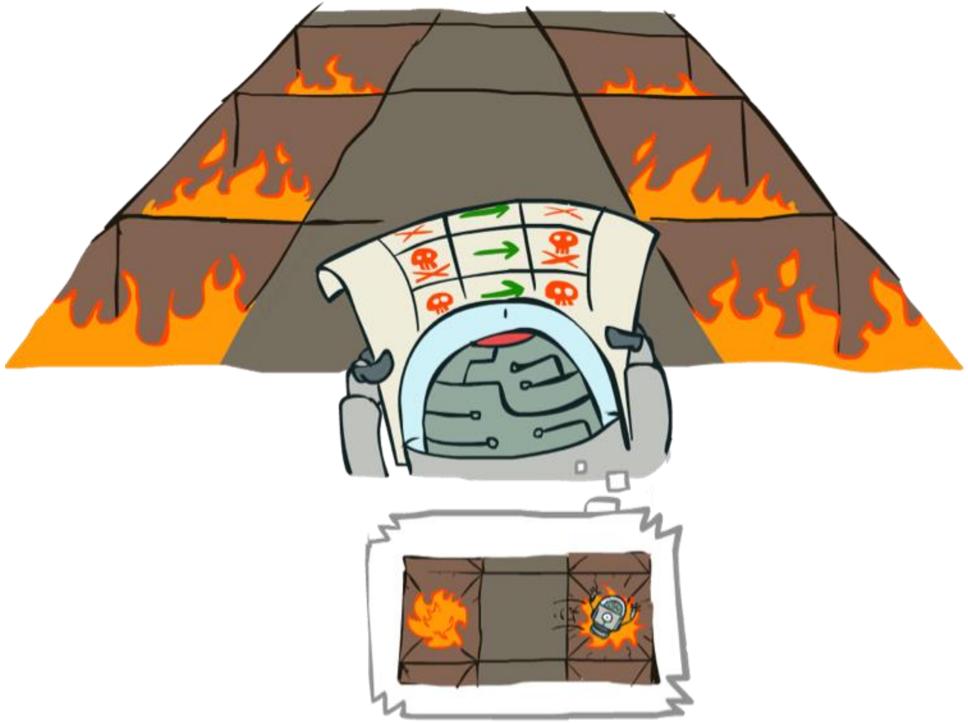
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

- Efficiency: O(S^2) per iteration

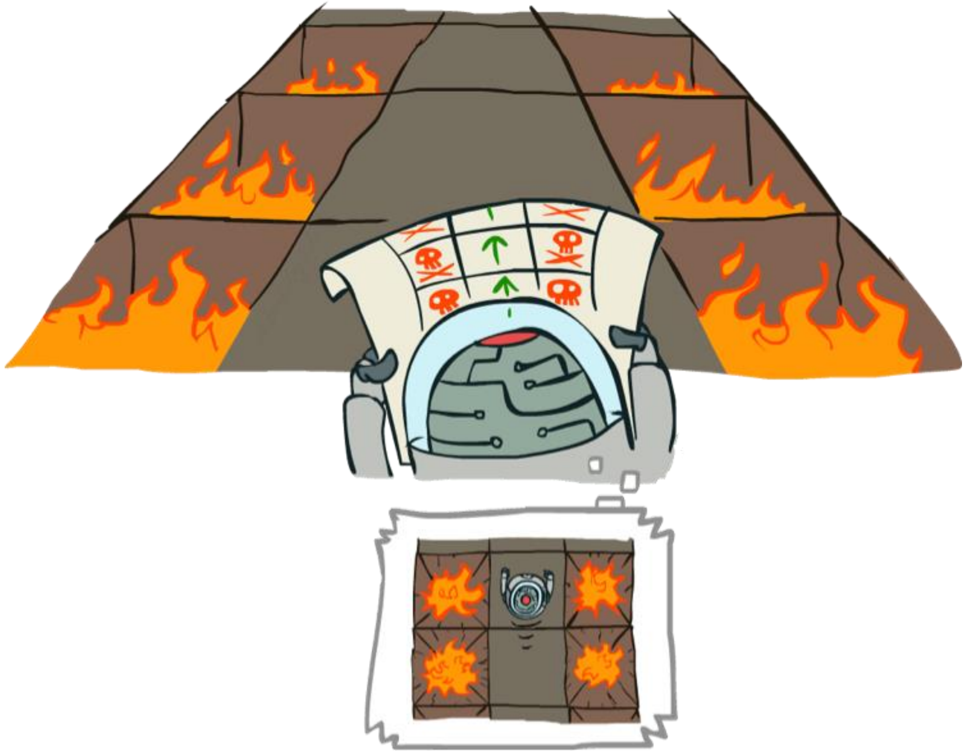
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with MATLAB (or your favorite linear system solver)

Example: Policy Evaluation

Always Go Right

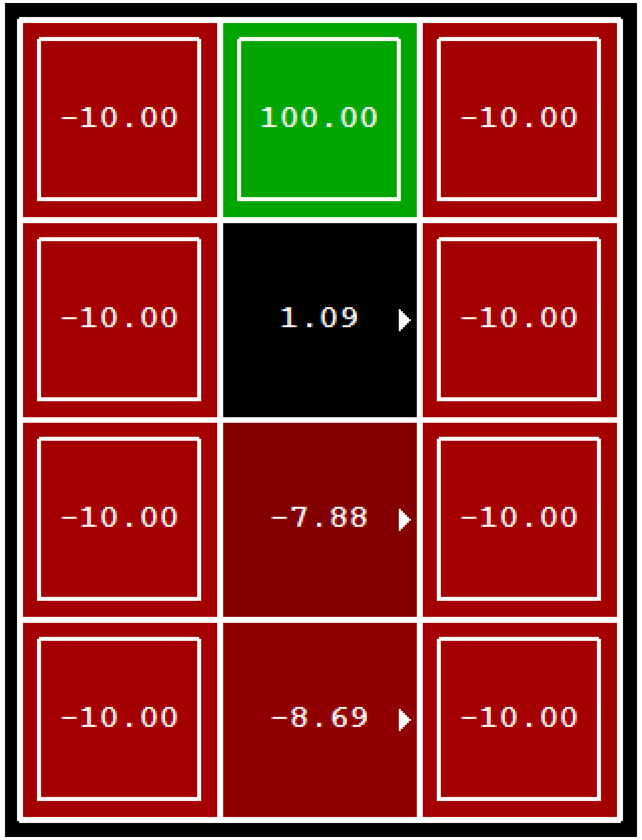


Always Go Forward



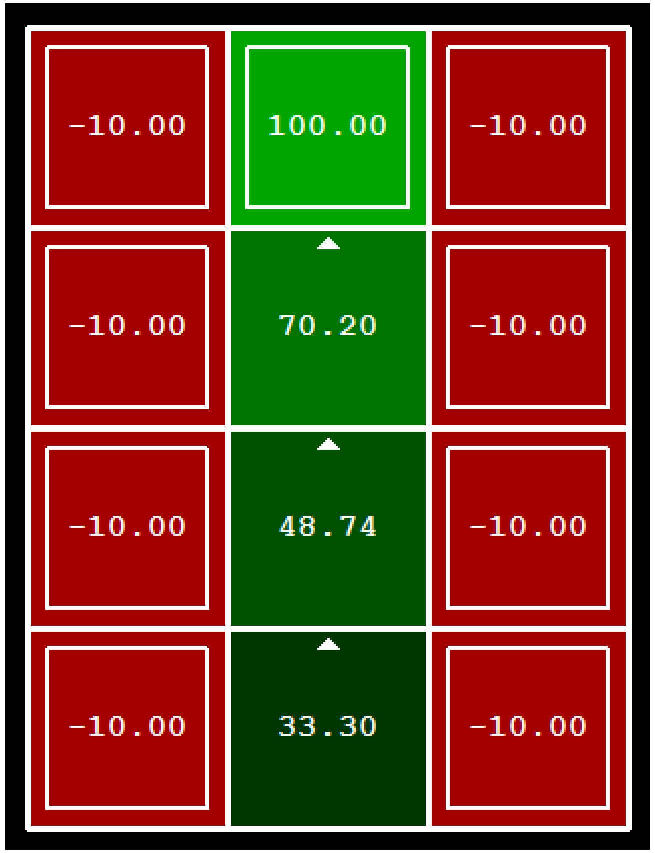
Example: Policy Evaluation 2

Always Go Right



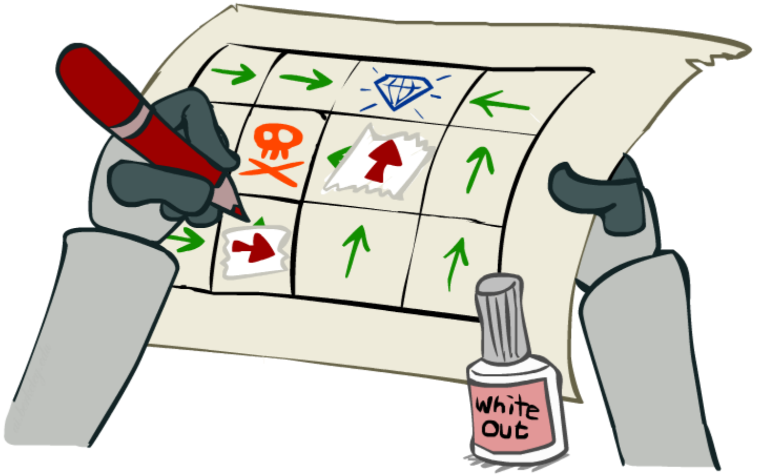
-10.00	100.00	-10.00
-10.00	1.09 ▸	-10.00
-10.00	-7.88 ▸	-10.00
-10.00	-8.69 ▸	-10.00

Always Go Forward



-10.00	100.00	-10.00
-10.00	▴ 70.20	-10.00
-10.00	▴ 48.74	-10.00
-10.00	▴ 33.30	-10.00

Policy Iteration



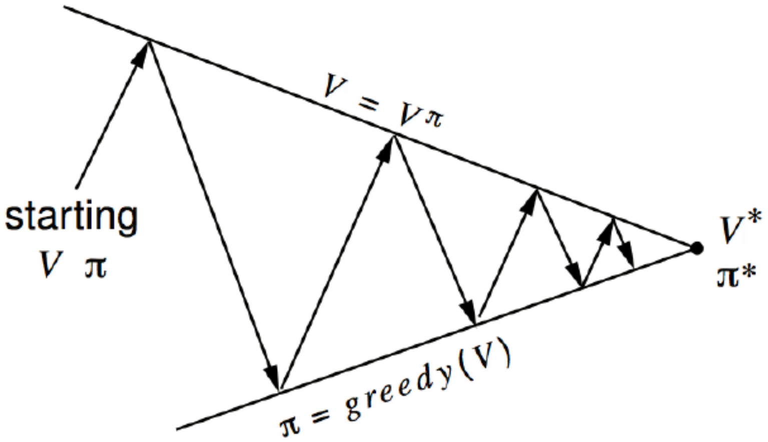
- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

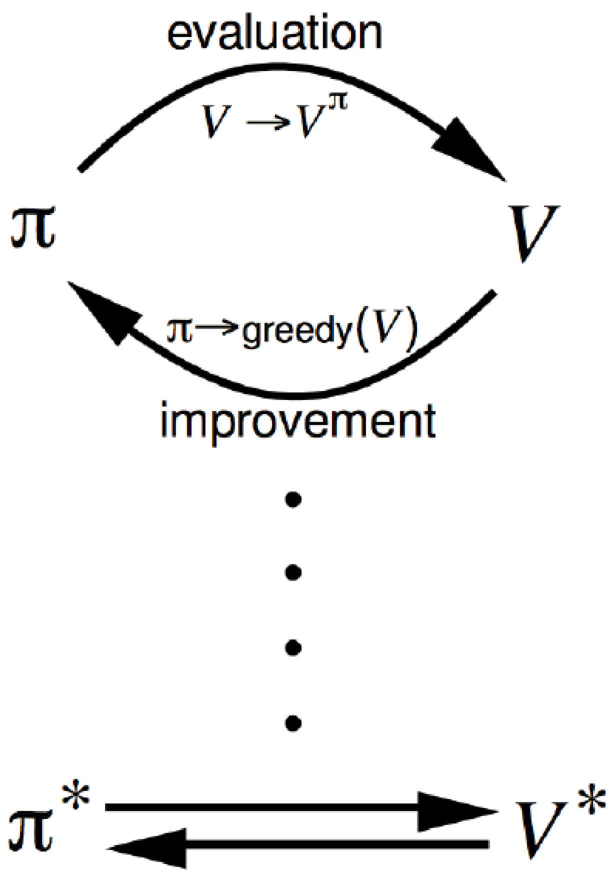
$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s') \left[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')\right]$$

- Improvement: For fixed values, get a better (why? exercise) policy using policy extraction
  - One-step look-ahead:

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

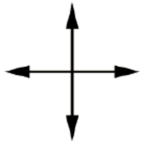
策略迭代





- ☐ 策略评估
  - 估计V^{π}
  - 迭代的评估策略
- ☐ 策略改进
  - 生成 $\pi' \geq \pi$
  - 贪心策略改进

举例：策略评估



▨	1	2	3
4	5	6	7
8	9	10	11
12	13	14	▨

- □ 非折扣MDP（$\gamma = 1$）
- □ 非终止状态：1, 2, ...,14
- □ 两个终止状态（灰色方格）
- □ 如果动作指向所有方格以外，则这一步不动
- □ 奖励均为-1，直到到达终止状态
- □ 智能体的初始策略为均匀随机策略

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 1/4$$

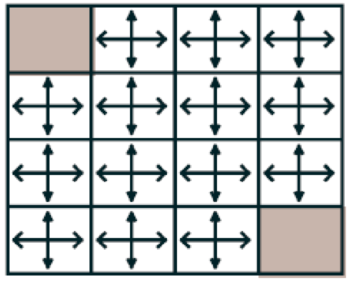
举例：策略评估

随机策略的 V_k

V_k对应的贪心策略

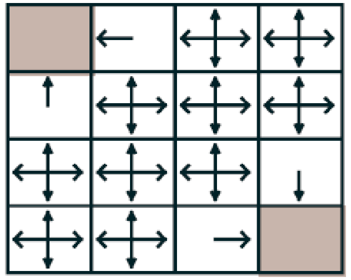
K=0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



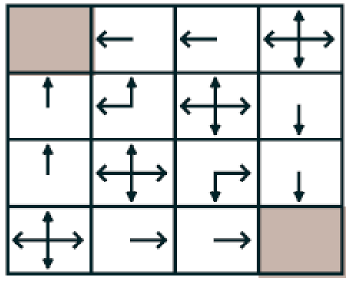
K=1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



K=2

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



举例：策略评估

随机策略的 V_k

V_k对应的贪心策略

K=3

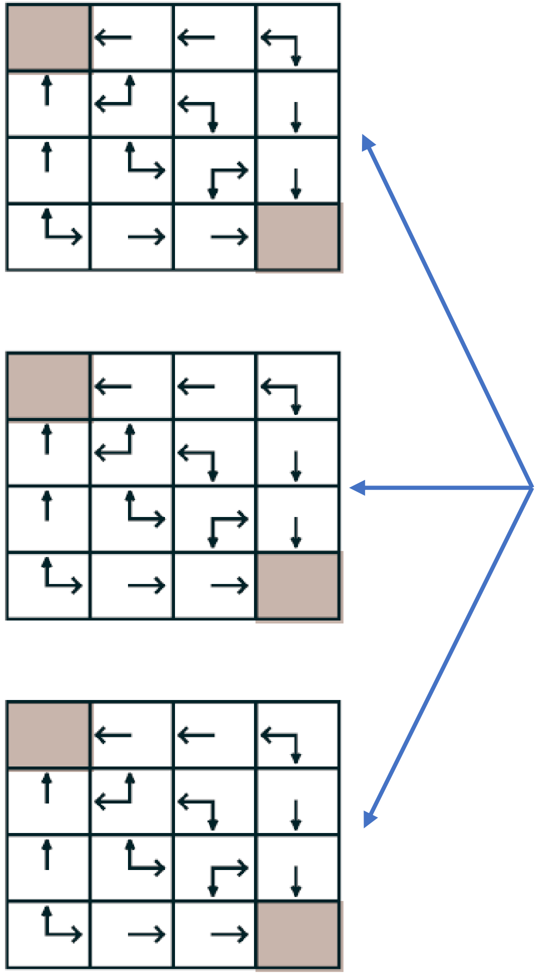
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

K=10

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

K=∞

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



$V := V^{\pi}$
最优策略

Summary of Two Methods for Solving MDPs

- Value iteration + policy extraction
  - Step 1: Value iteration: calculate values for all states by running one ply of the Bellman equations using values from previous iteration **until convergence**
  - Step 2: Policy extraction: compute policy by running one ply of the Bellman equations using values from value iteration
- Policy iteration
  - Step 1: Policy evaluation: calculate values for some fixed policy (not optimal values!) **until convergence**
  - Step 2: Policy improvement: update policy by running one ply of the Bellman equations using values from policy evaluation
  - **Repeat** steps until policy converges

Value Iteration vs. Policy Iteration

- 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

价值迭代 vs. 策略迭代

价值迭代

1. 对每个状态s，初始化 $V(s) = 0$
2. 重复以下过程直到收敛 {
   对每个状态，更新

$$V_{k+1}(s) = \max_{a} \sum_{s'} P_{s,a}(s')\,[R(s,a,s') + \gamma V_k(s')]$$

}

策略迭代

1. 随机初始化策略 π
2. 重复以下过程直到收敛 {
   a) 让 $V := V^{\pi}$
   b) 对每个状态，更新

$$\pi(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{s,a}(s')[R(s,a,s') + \gamma V(s')]$$

}

备注：

1. 价值迭代是贪心更新法
2. 策略迭代中，用Bellman等式更新价值函数代价很大
3. 对于空间较小的MDP，策略迭代通常很快收敛
4. 对于空间较大的MDP，价值迭代更实用（效率更高）
5. 如果没有状态转移循环，最好使用价值迭代

基于模型的强化学习

学习一个MDP模型

- 目前我们关注在给出一个已知MDP模型后：（也就是说，状态转移$P_{sa}(s')$和奖励函数$R(s)$明确给定后）
  - 计算最优价值函数
  - 学习最优策略
- 在实际问题中，状态转移和奖励函数一般不是明确给出的
  - 比如，我们只看到了一些episodes

Episode1： $s_0^{(1)} \xrightarrow{a_0^{(1)},R(s_0)^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)},R(s_1)^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)},R(s_2)^{(1)}} s_3^{(1)} \cdots s_T^{(1)}$

Episode2： $s_0^{(2)} \xrightarrow{a_0^{(2)},R(s_0)^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)},R(s_1)^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)},R(s_2)^{(2)}} s_3^{(2)} \cdots s_T^{(2)}$

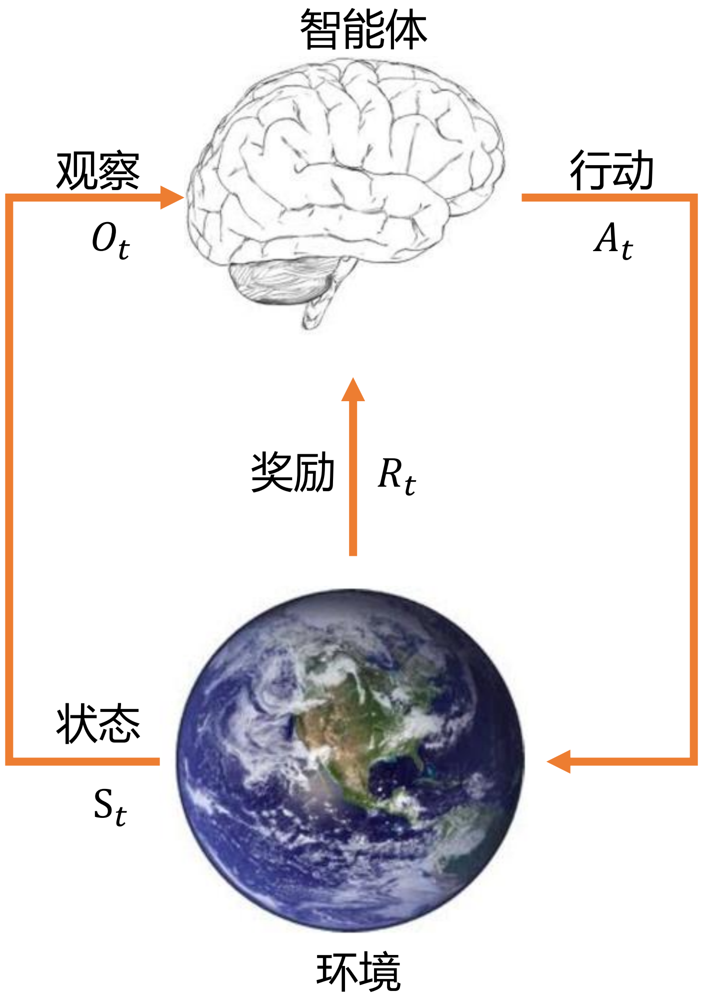
Recall: 强化学习系统要素

- 环境的模型（Model）用于模拟环境的行为
  - 预测下一个状态

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

  - 预测下一个（immediate 立即）奖励

$$\mathcal{R}_{s}^{a} = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$



学习一个MDP模型

Episode1：　$s_0^{(1)} \xrightarrow{a_0^{(1)},R(s_0)^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)},R(s_1)^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)},R(s_2)^{(1)}} s_3^{(1)} \cdots s_T^{(1)}$

Episode2：　$s_0^{(2)} \xrightarrow{a_0^{(2)},R(s_0)^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)},R(s_1)^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)},R(s_2)^{(2)}} s_3^{(2)} \cdots s_T^{(2)}$

⋮　　　　　　　　⋮

□ 从“经验”中学习一个MDP模型

- 学习状态转移概率$P_{sa}(s')$

$$P_{sa}(s') = \frac{\text{在}s\text{下采取动作}a\text{并转移到}s'\text{的次数}}{\text{在}s\text{下采取动作}a\text{的次数}}$$

- 学习奖励函数$R(s)$，也就是立即奖赏期望

$$R(s) = \text{average}\{R(s)^{(i)}\}$$

学习模型&优化策略

- 算法

1. 随机初始化策略π

2. 重复以下过程直到收敛 {

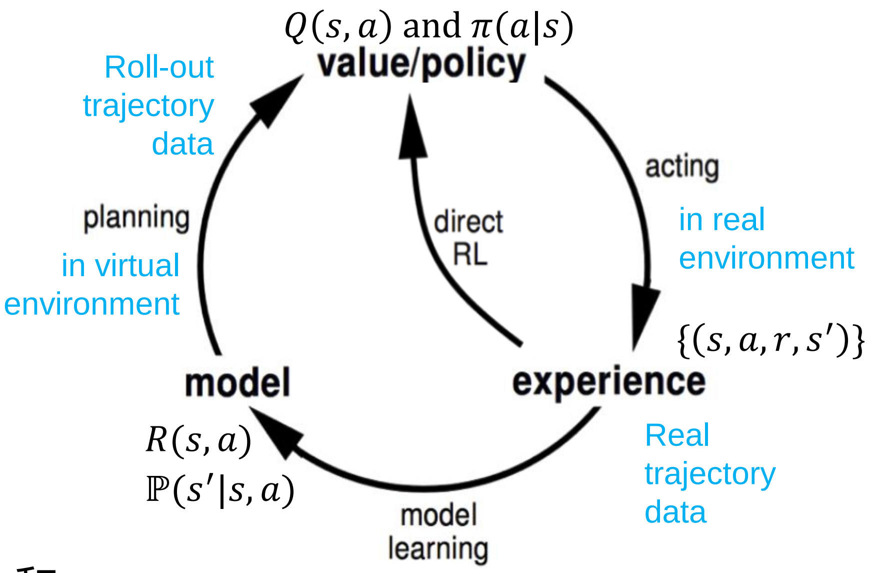
   a) 在MDP中执行π，收集经验数据

   b) 使用MDP中的累积经验更新对P_{sa}和R的估计

   c) 利用对P_{sa}和R的估计执行价值迭代，得到新的估计价值函数V

   d) 根据V更新策略π为贪心策略

}



学习一个MDP模型

- 在实际问题中，状态转移和奖励函数一般不是明确给出的
  - 比如，我们只看到了一些episodes

Episode1：$s_0^{(1)} \xrightarrow{a_0^{(1)}, R(s_0)^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}, R(s_1)^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}, R(s_2)^{(1)}} s_3^{(1)} \cdots s_T^{(1)}$

Episode2：$s_0^{(2)} \xrightarrow{a_0^{(2)}, R(s_0)^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}, R(s_1)^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}, R(s_2)^{(2)}} s_3^{(2)} \cdots s_T^{(2)}$

- 另一种解决方式是不学习MDP，从经验中直接学习价值函数和策略
  - 也就是模型无关的强化学习（Model-free Reinforcement Learning）

马尔可夫决策过程总结

- MDP由一个五元组构成 $(S, A, \{P_{sa}\}, \gamma, R)$，其中状态转移$P$和奖励函数$R$构成了动态系统

- 动态系统和策略交互的占用度量

$$\rho^{\pi}(s,a) = \sum_{t=0}^{T} \gamma^{t}\, \mathbb{P}(s_t = s, a_t = a | s_0, \pi)$$

- 一个白盒环境给定的情况下，可用动态规划的方法求解最优策略
  - 价值迭代和策略迭代

- 如果环境是黑盒的，可以根据统计信息来拟合出动态环境P和R，然后做动态规划求解最优策略

THANK YOU