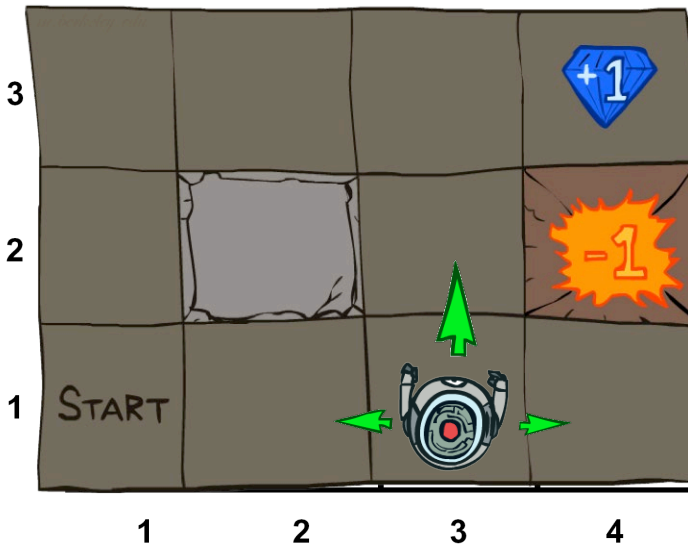
MDP 第二部分

Example: Grid World

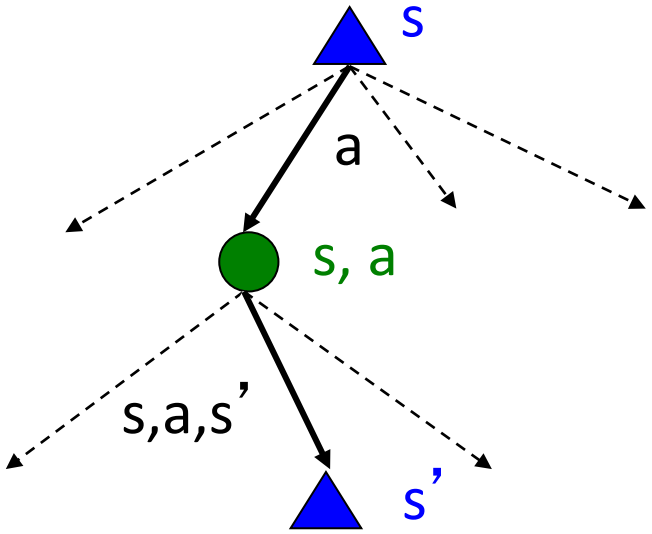


- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time,

    the action North takes the agent North
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of (discounted) rewards

Recap: MDPs

- Markov decision processes:
  - States S
  - Actions A
  - Transitions P(s'|s,a) (or T(s,a,s'))
  - Rewards R(s,a,s') (and discount γ)
  - Start state s_0



- Quantities:
  - Policy = map of states to actions
  - Utility = sum of discounted rewards
  - Values = expected future utility from a state (max node)
  - Q-Values = expected future utility from a q-state (chance node)

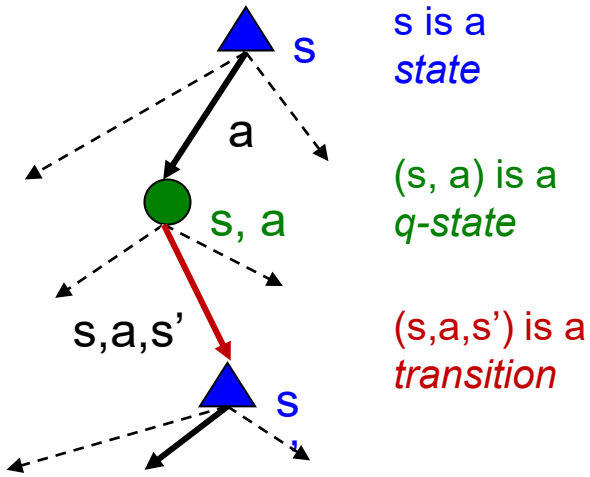
Recap: Optimal Quantities

- 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

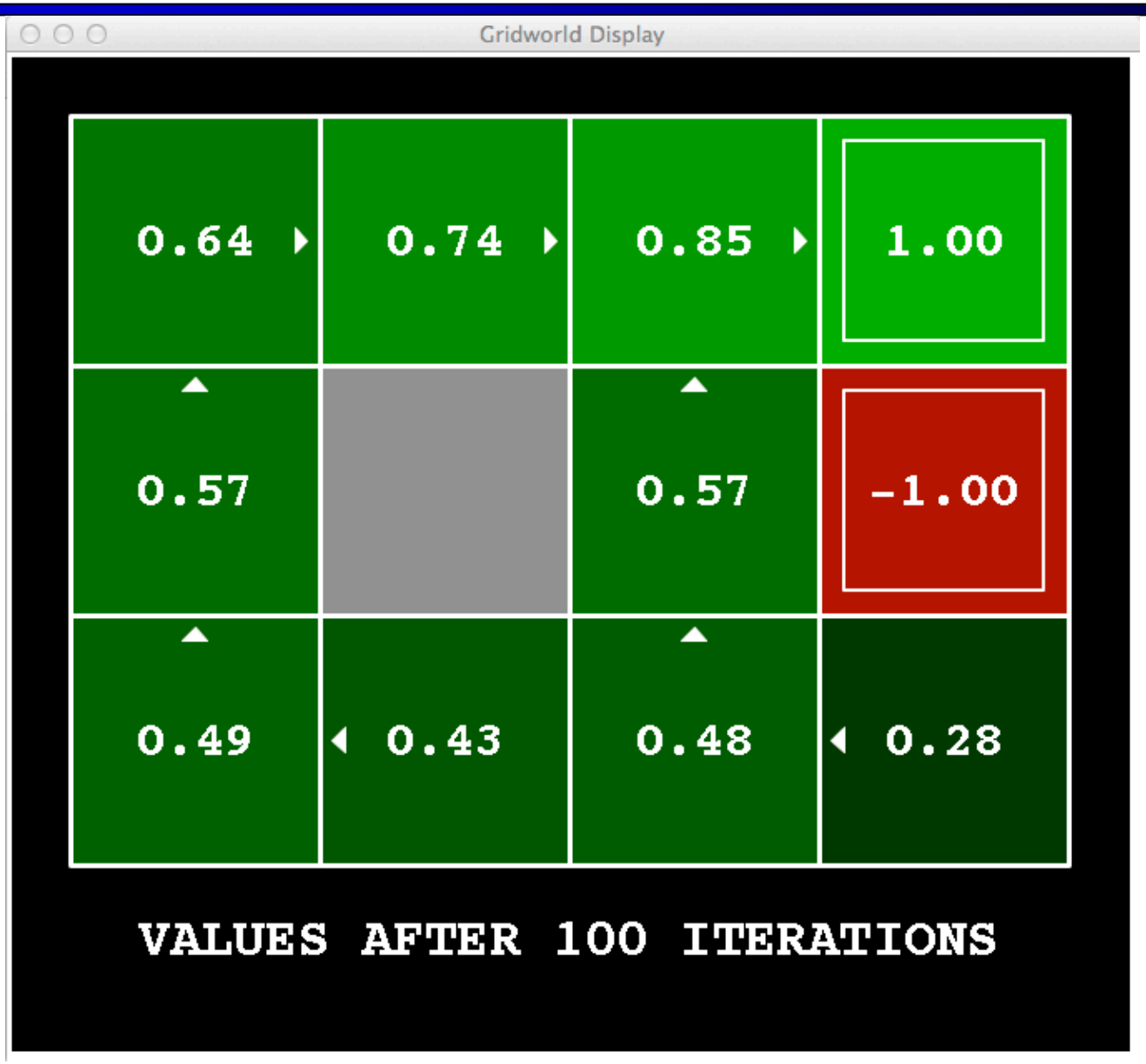


- The optimal policy:

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

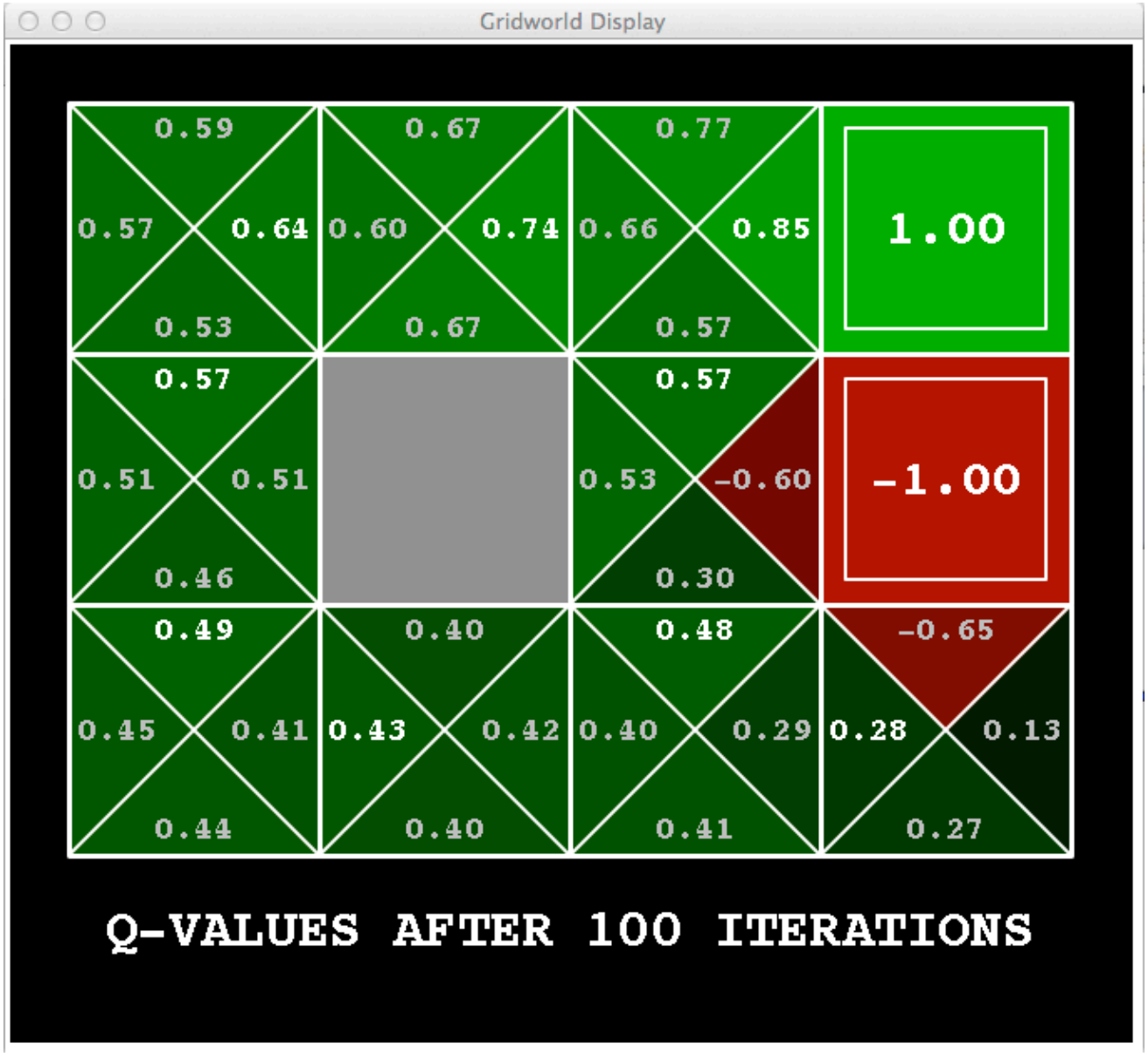
[Demo: gridworld values (L9D1)]

Snapshot of Gridworld - V Values



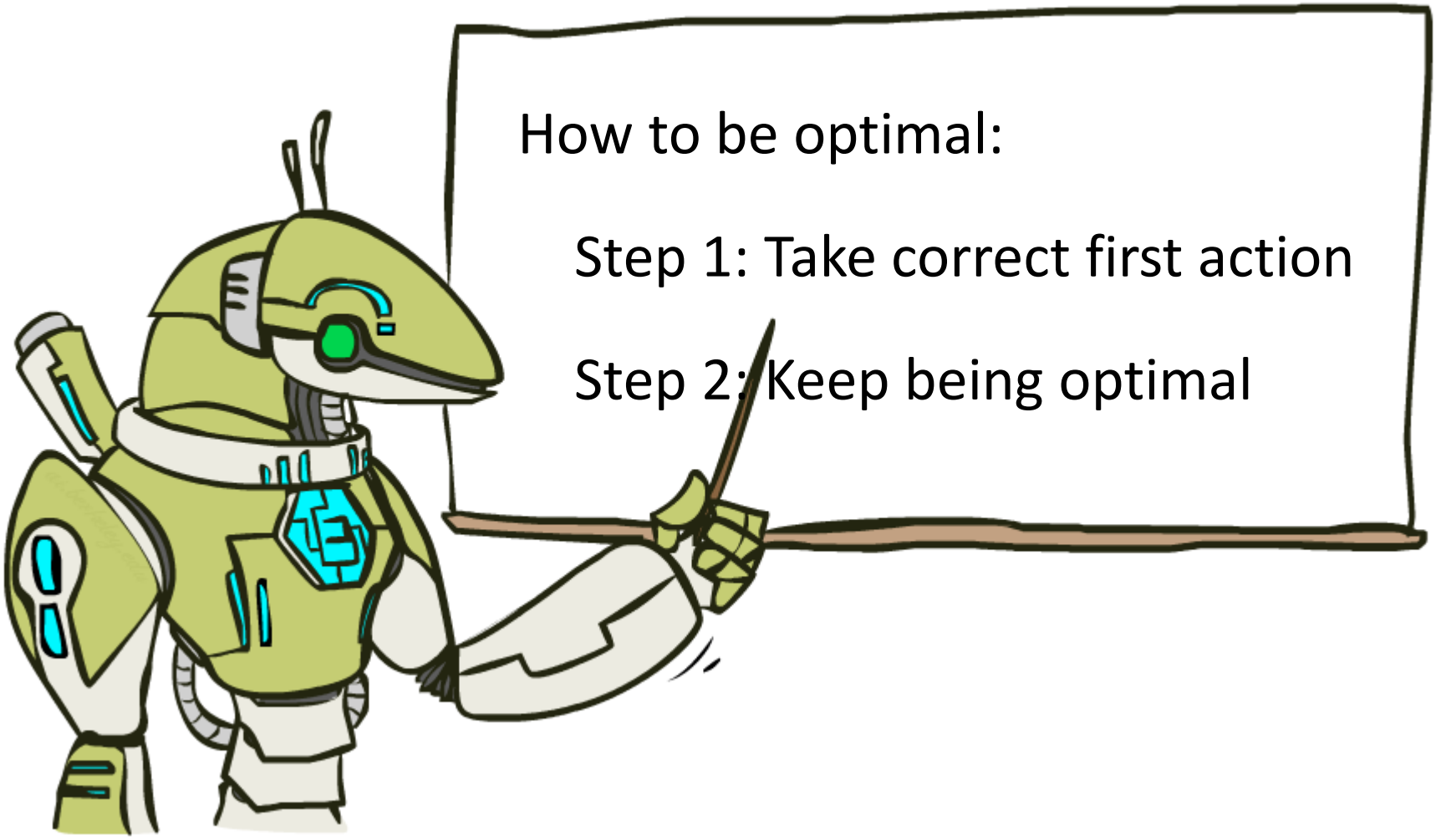
Noise = 0.2
Discount = 0.9
Living reward = 0

Snapshot of Gridworld - Q Values



Noise = 0.2
Discount = 0.9
Living reward = 0

The Bellman Equations



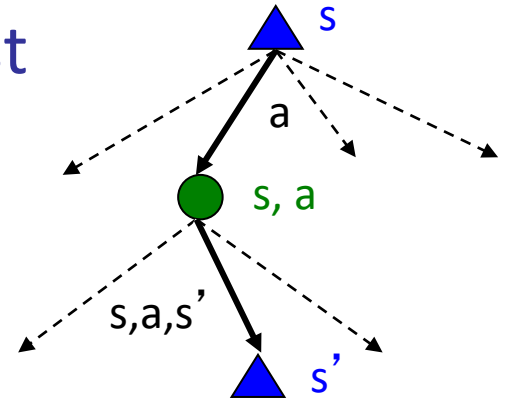
The Bellman Equations

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

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

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

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

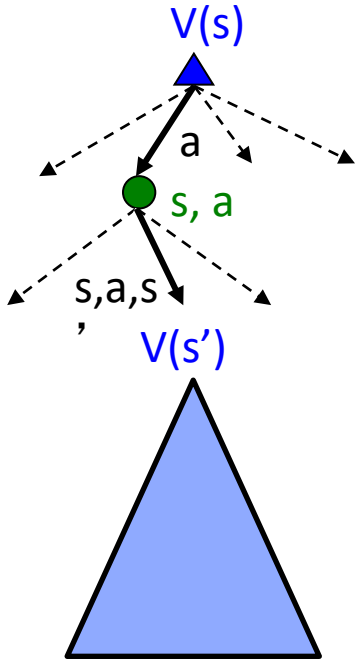


- These are the Bellman equations, and they characterize optimal values in a way we'll use over and over

Recap: Value Iteration（值迭代）

- Bellman equations characterize the optimal values:

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



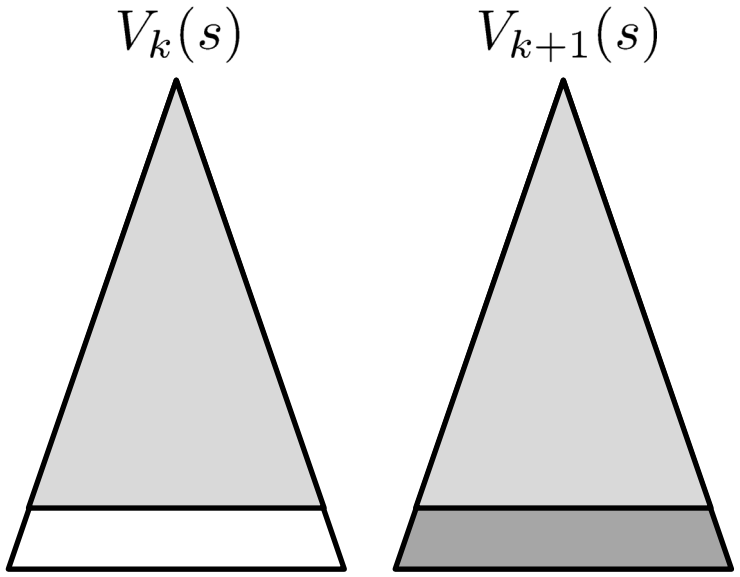
- Value iteration computes them:

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

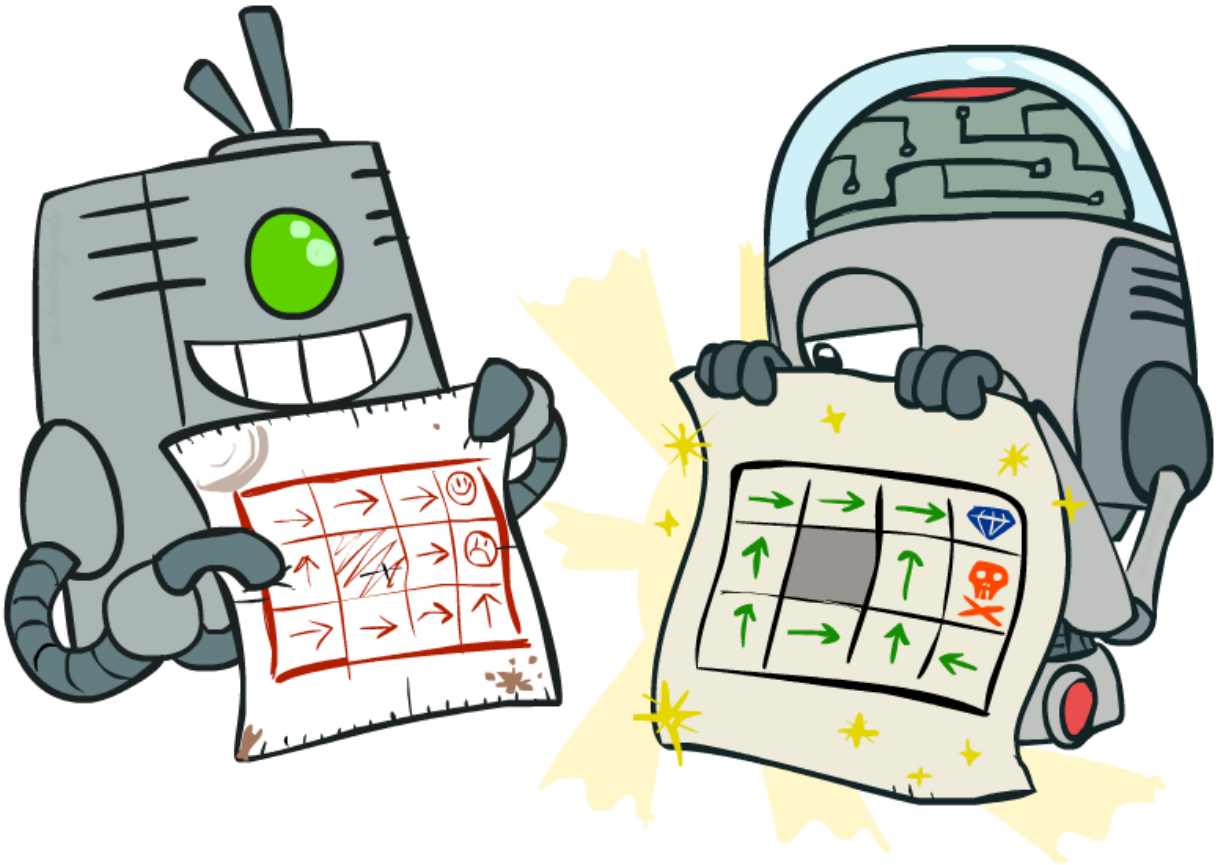
- Value iteration is just a fixed point solution method
  - … though the V_k vectors are also interpretable as time-limited values

Convergence* （收敛）

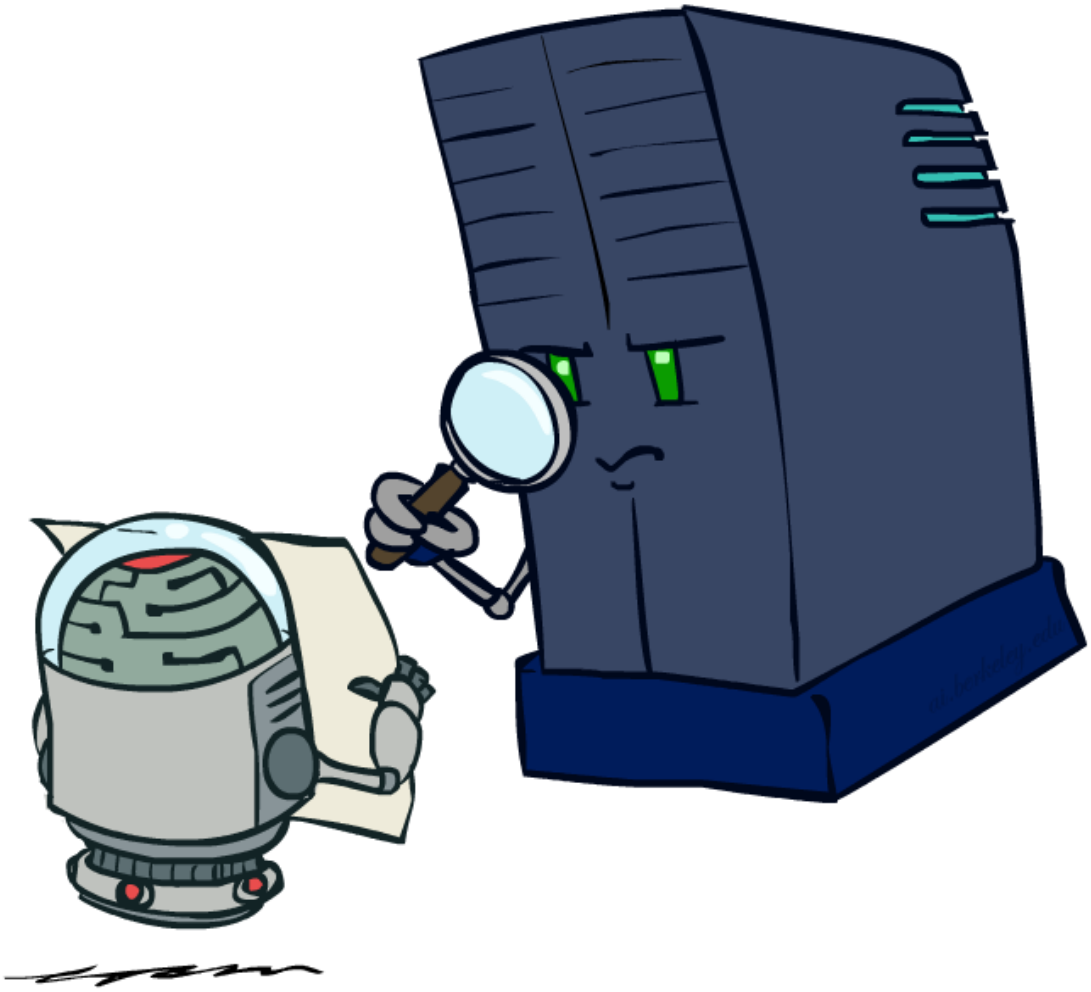
- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - 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



基于策略的求解MDP方法
Policy Methods

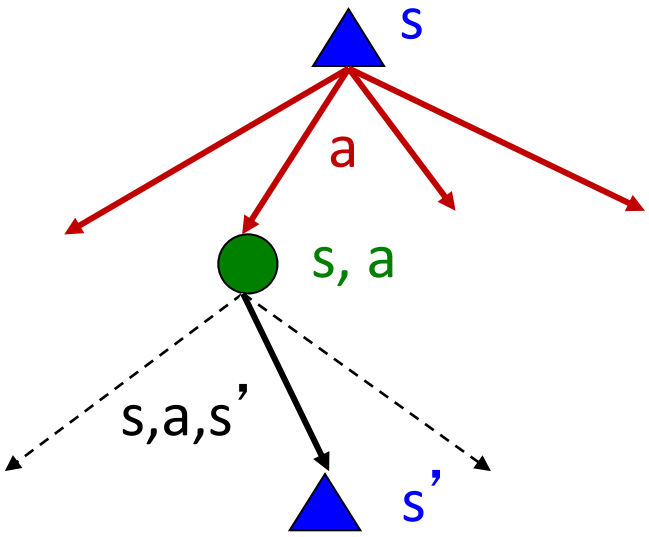


策略评价

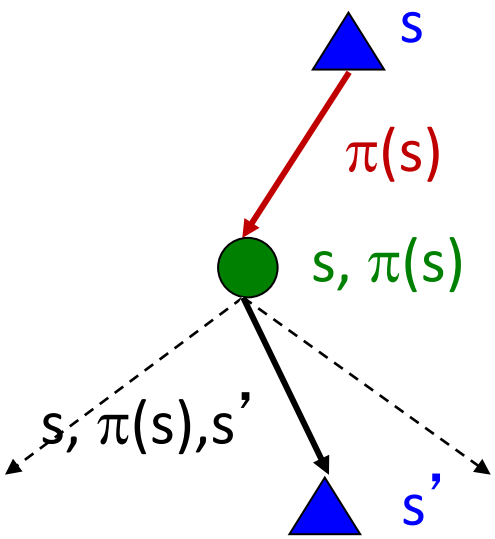


固定的策略

采取最优行动



按照策略 π 选择行动



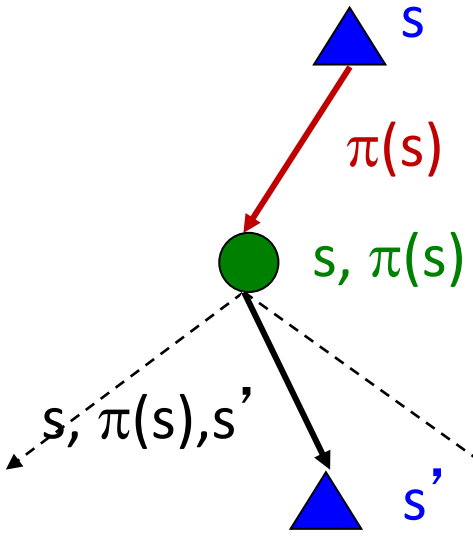
- 期望最大值搜索树在所有行动分支里选择最大功效值
- 如果使用一个给定的策略 π(s)，那么这个搜索树变得比以前要简单 – 每个状态节点只有一个分支
  - … 所以搜索树的状态节点值的计算将取决于我们所固定（使用）的那个策略

用来评价一个固定的策略的功效值

- 当给定一个策略（通常不是最优的）时，如何计算一个状态 s 的功效值

- 给定策略 π 下，定义一个状态 s 的功效值为:

  $V^{\pi}(s)$ = 从状态 s 开始，按照策略 π 执行，所获得的期望折扣奖赏值的总计

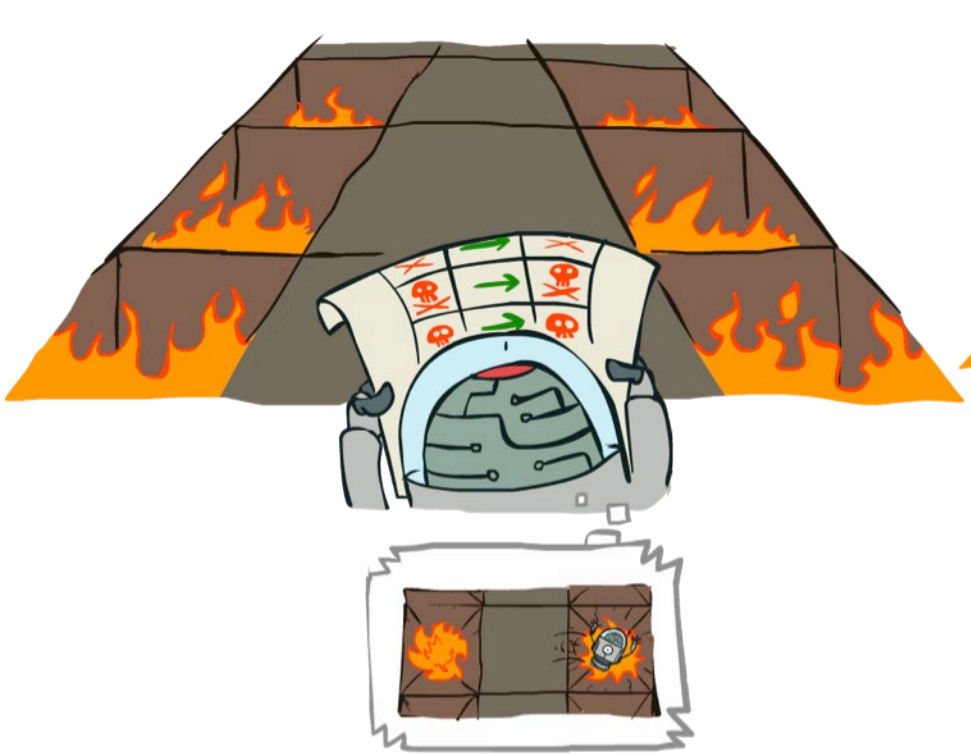


- 迭代关系（基于 Bellman 公式的一步计算）:

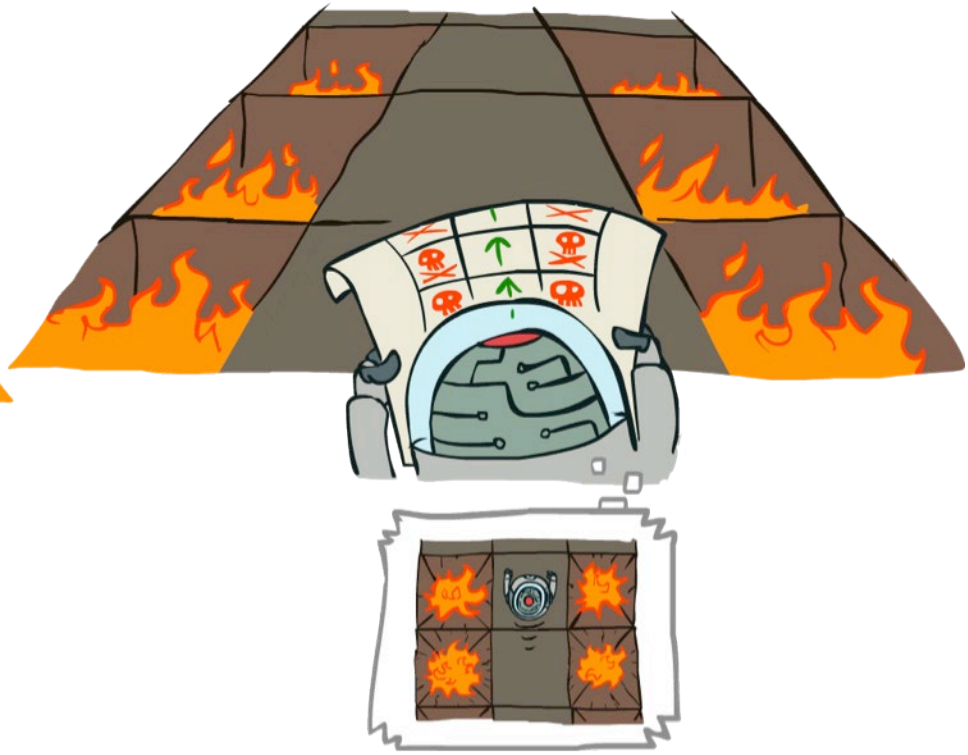
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

举例：策略评价

Always Go Right



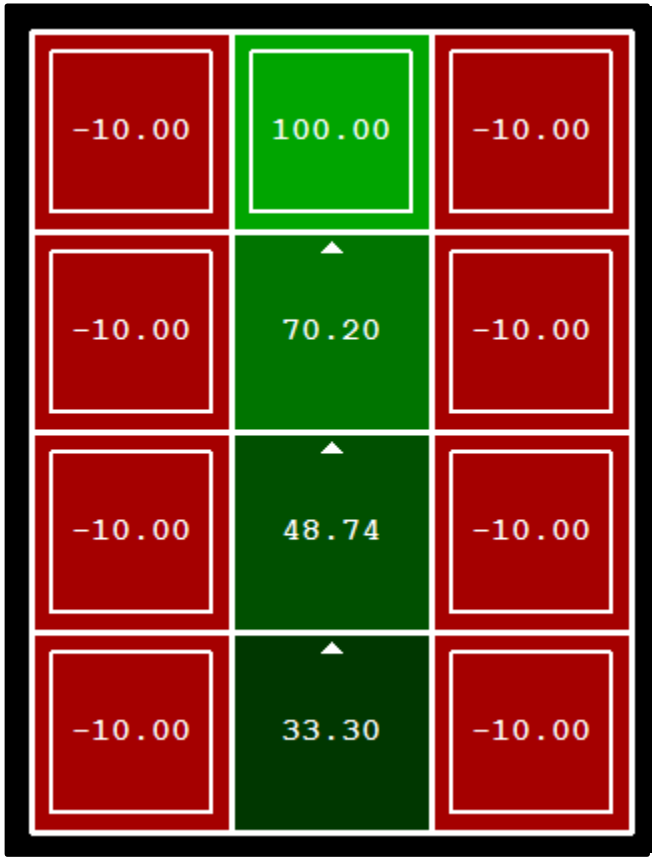
Always Go Forward



举例：策略评价

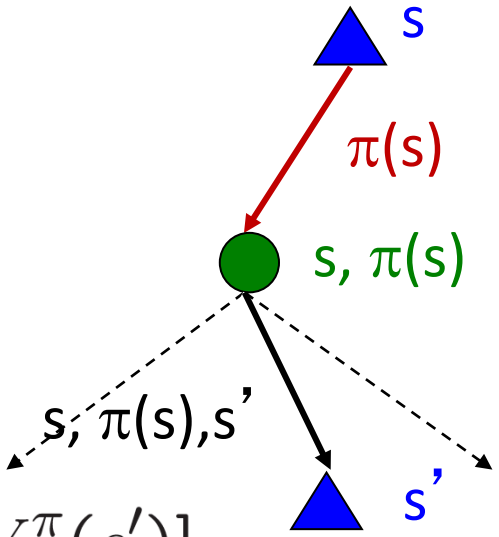
Always Go Right

Always Go Forward



策略评价

- 给定一个策略π，如何计算 V-值?
- 思路 1：基于 Bellman 方程的赋值迭代更新



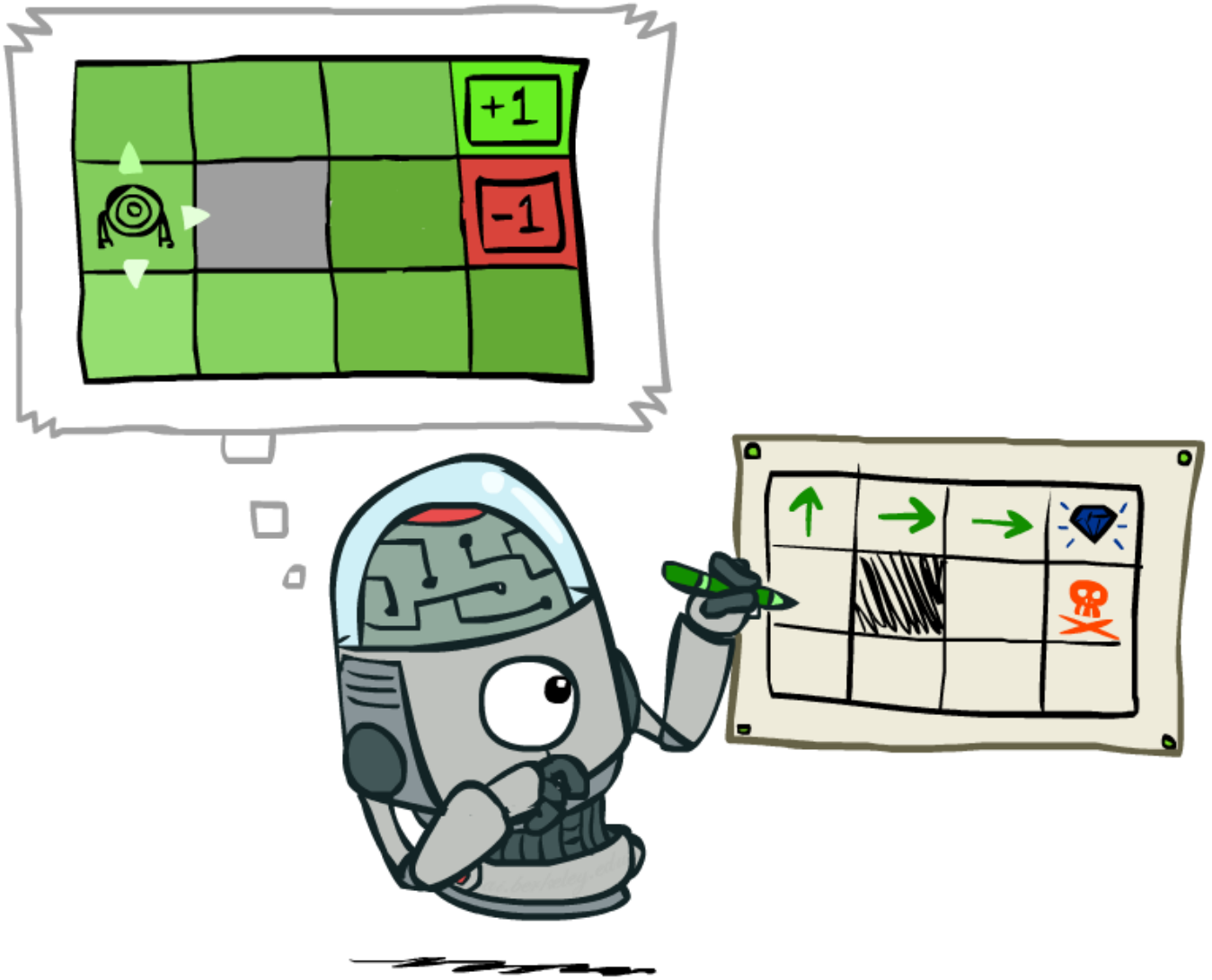
$$V_0^{\pi}(s) = 0$$

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

  - 效率：$O(S^2)$ 每步迭代

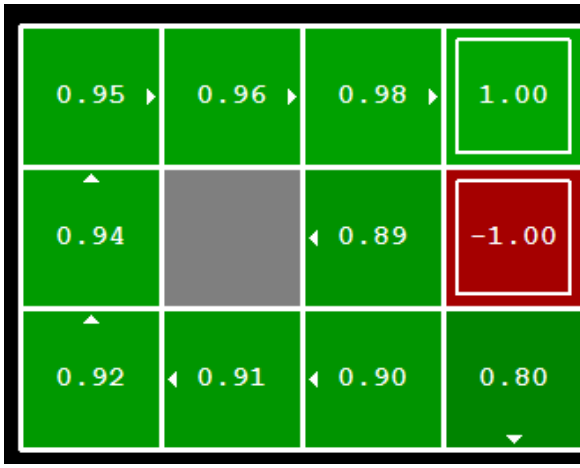
- 思路 2：由于没有了“最大符号”，Bellman 方程变为了一个线性系统

策略提取 Policy Extraction



从状态值计算行动

- 假设我们已有了每个状态的最优值 V*(s)
- 那么我们如何选择行动?
  - 不是很明显!
- 需要计算一步长的 mini-expectimax:

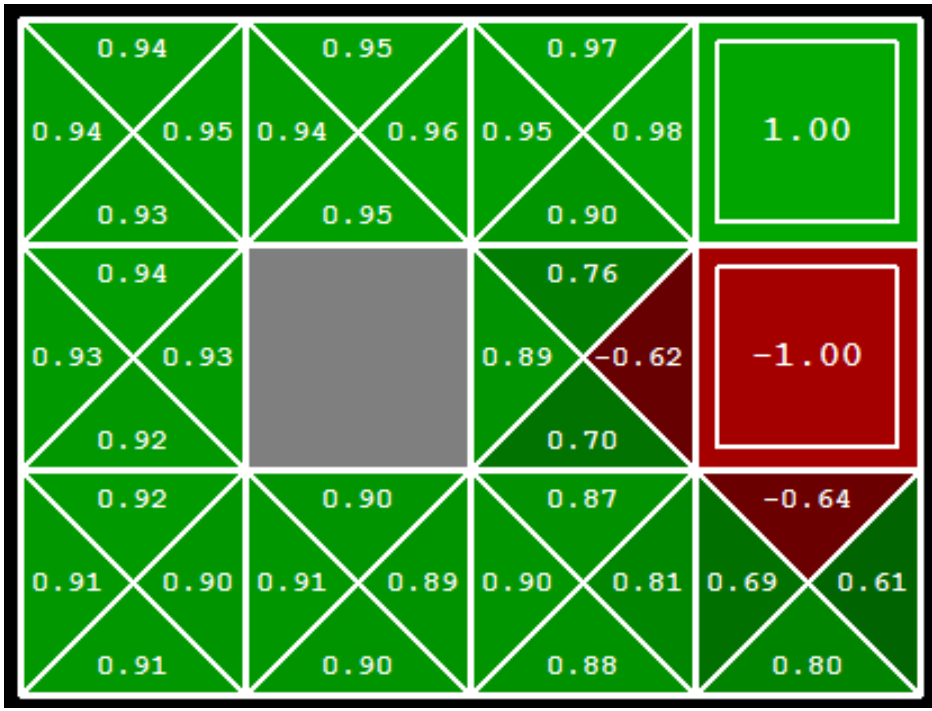


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- 这叫做 策略提取，通过计算期望最大值，间接地获得最优行动选择

从Q-值计算行动

- 假设我们已有了最优的 q-values:

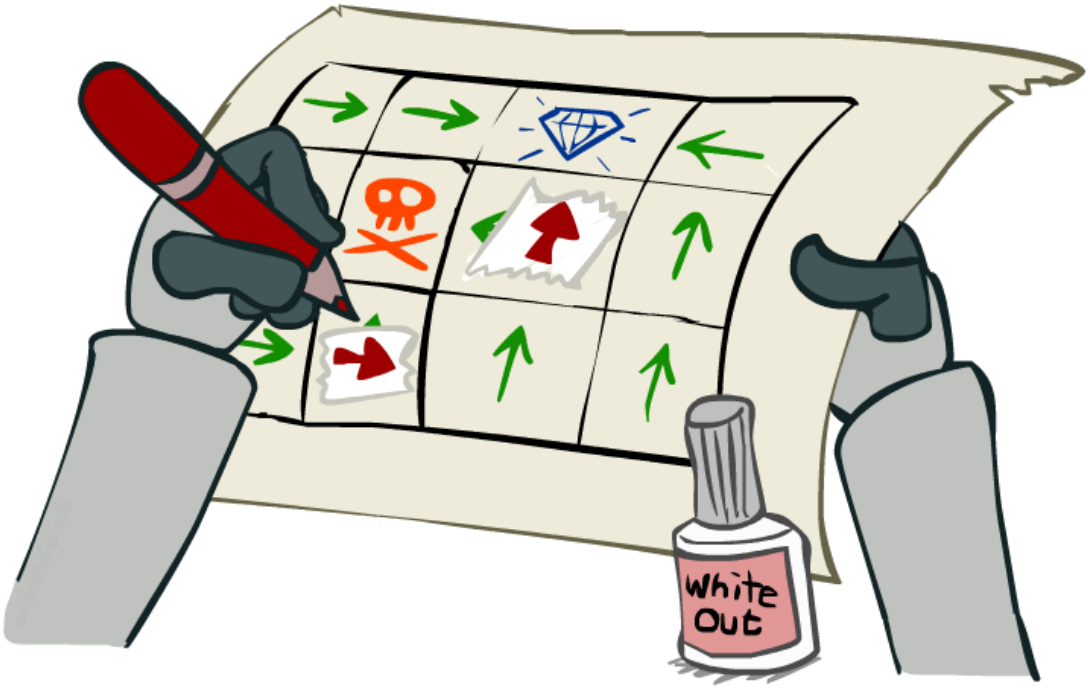


- 现在我们如何挑选行动?
  - 此时变得很简单!

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

- 因此：为每个状态位置选择最优行动从Q-值中比在状态值中更容易!

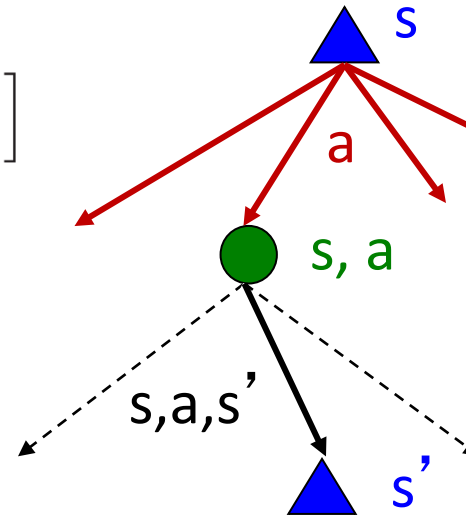
策略迭代法 Policy Iteration



状态赋值迭代方法的缺点

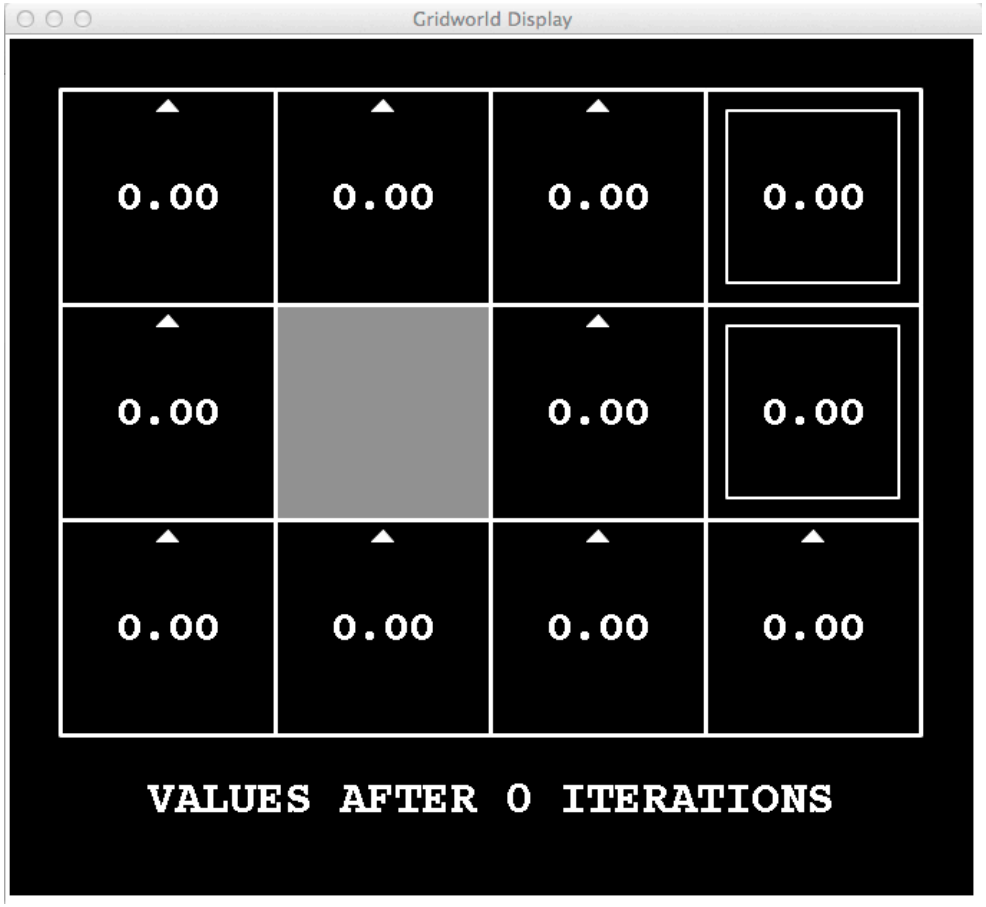
- 赋值迭代实现的是基于 Bellman 方程的更新公式：

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



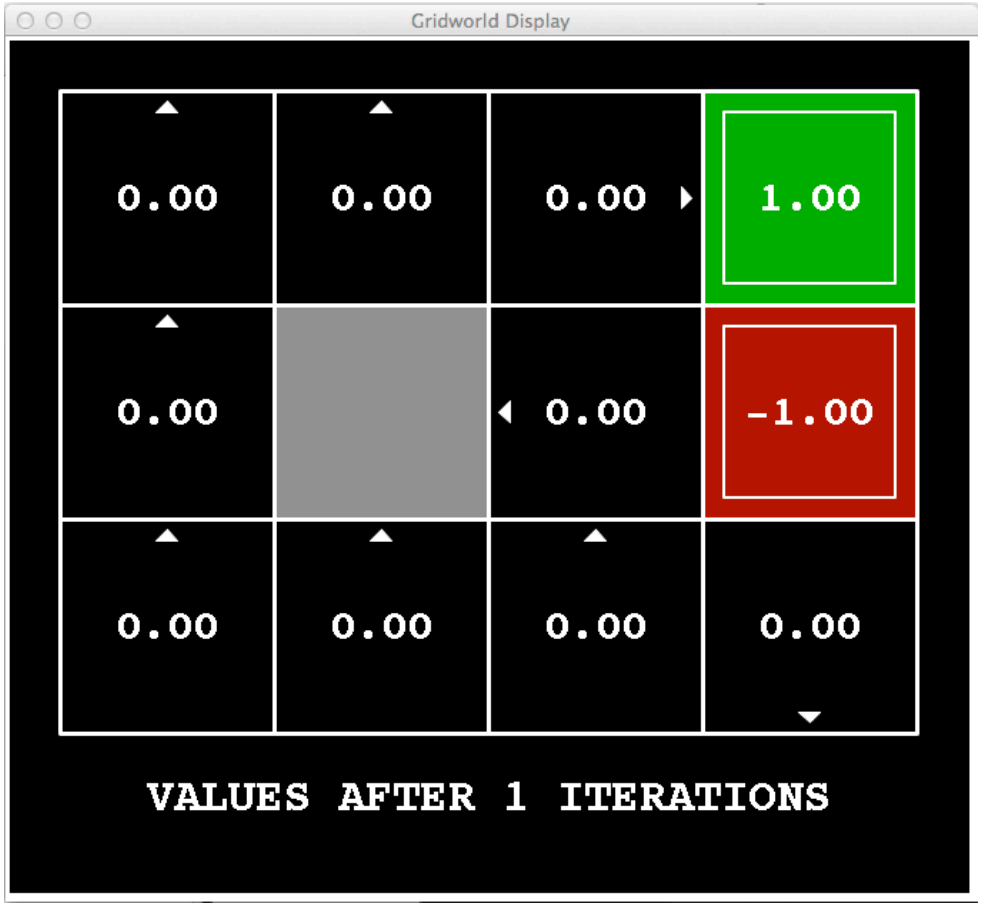
- 问题 1：很慢 – 每次迭代的复杂度是 $O(S^2A)$
- 问题 2：在每个位置（状态）的 “最大”选项很少改变
- 问题 3：策略 policy 通常比状态值收敛的更快

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

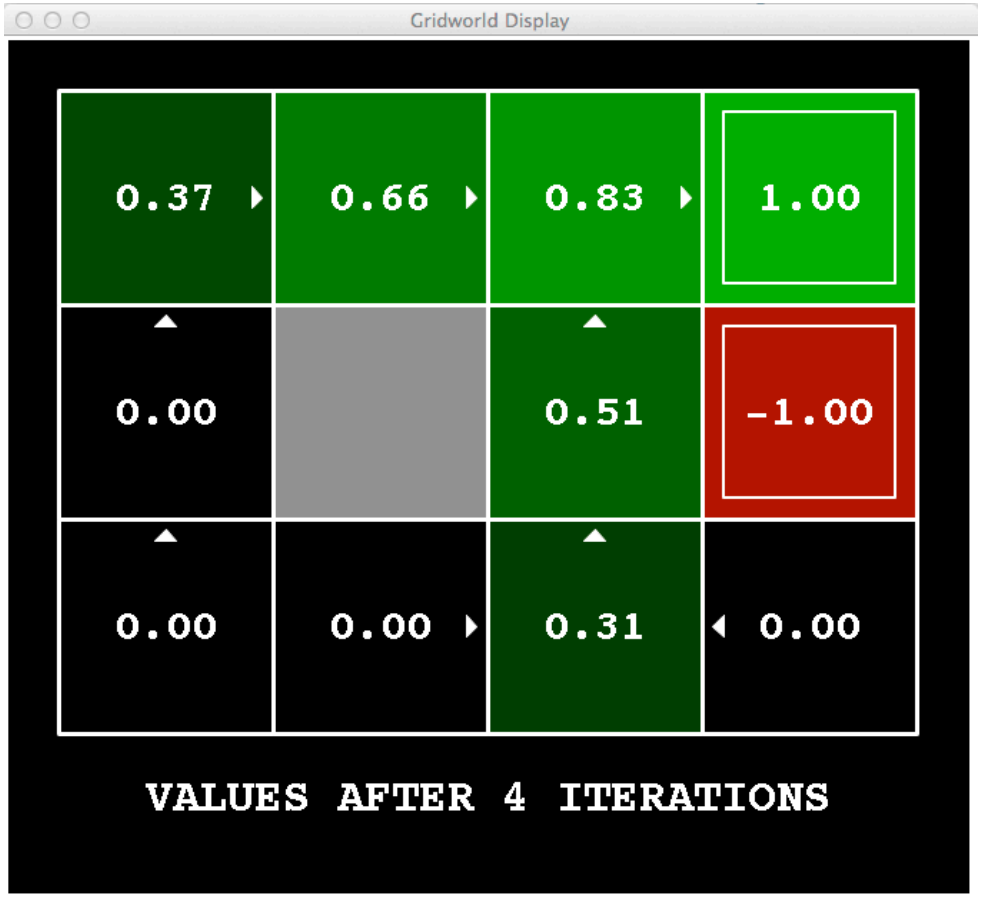
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

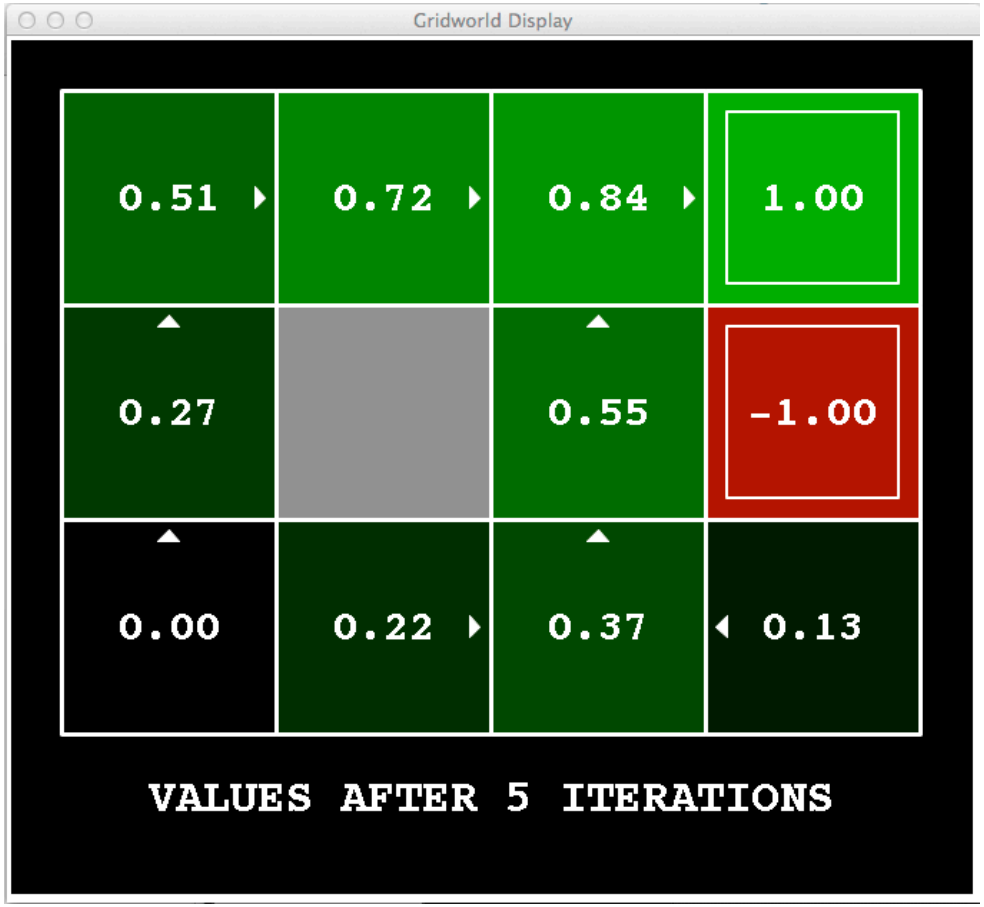
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



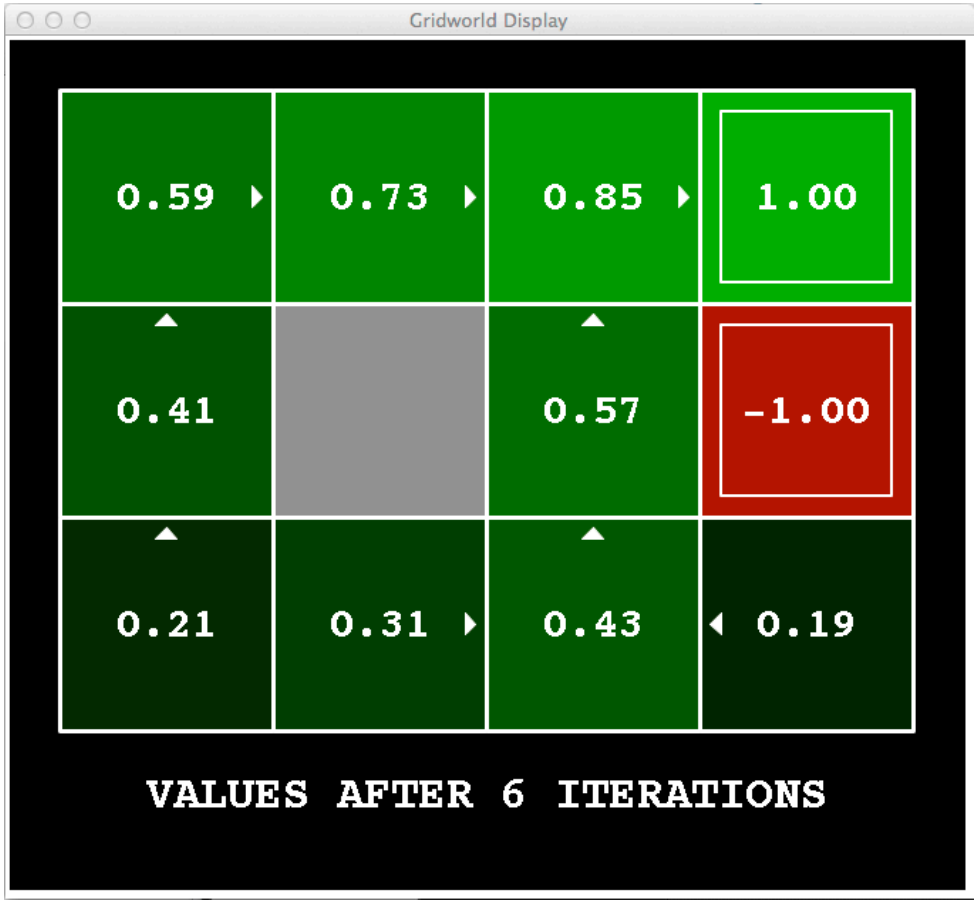
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



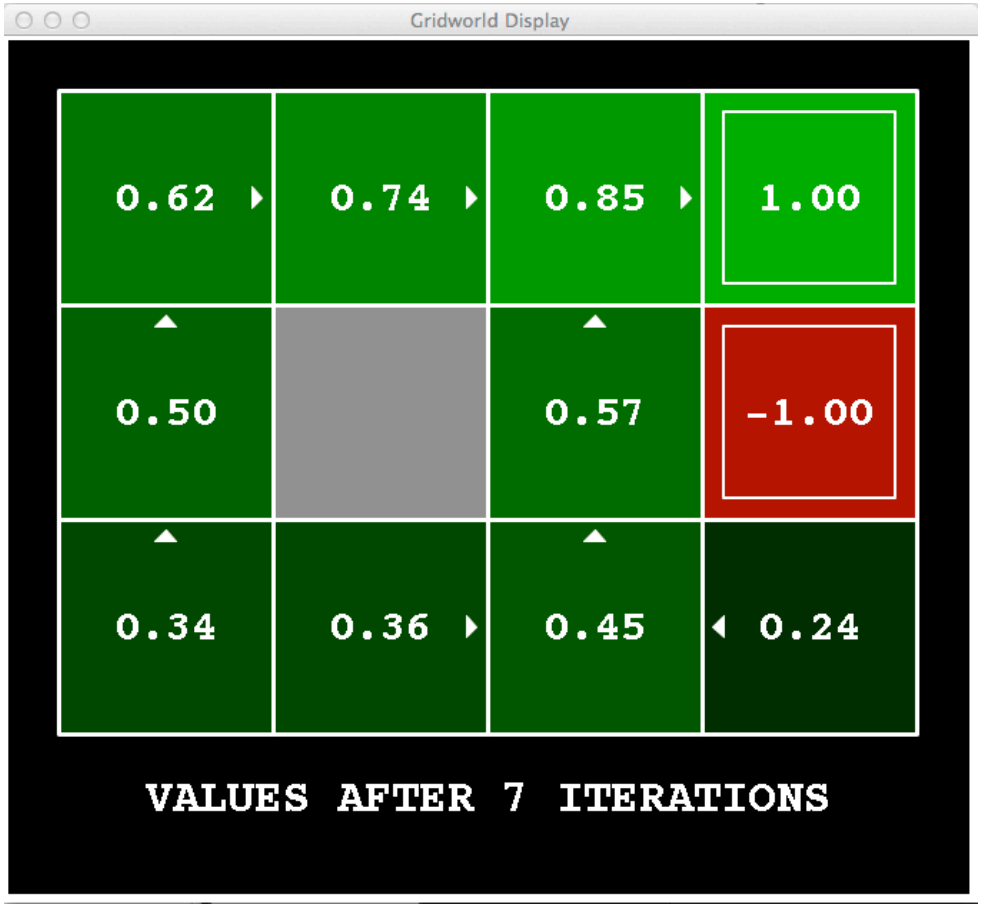
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



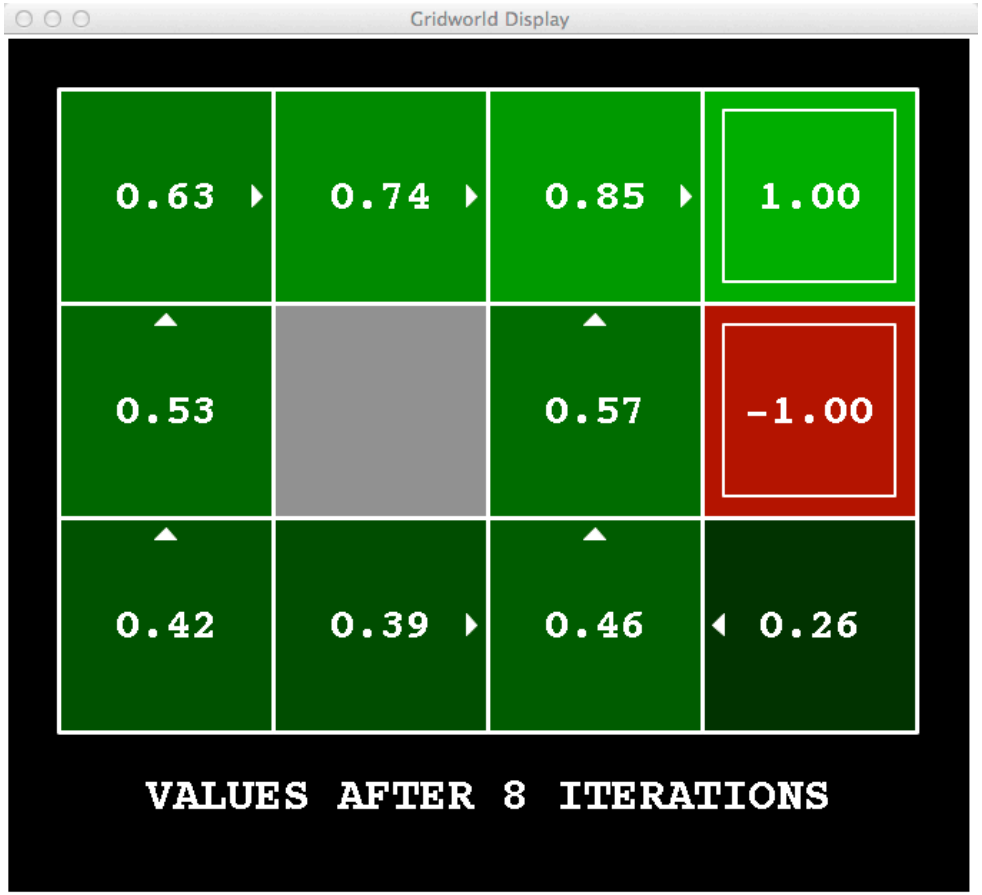
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



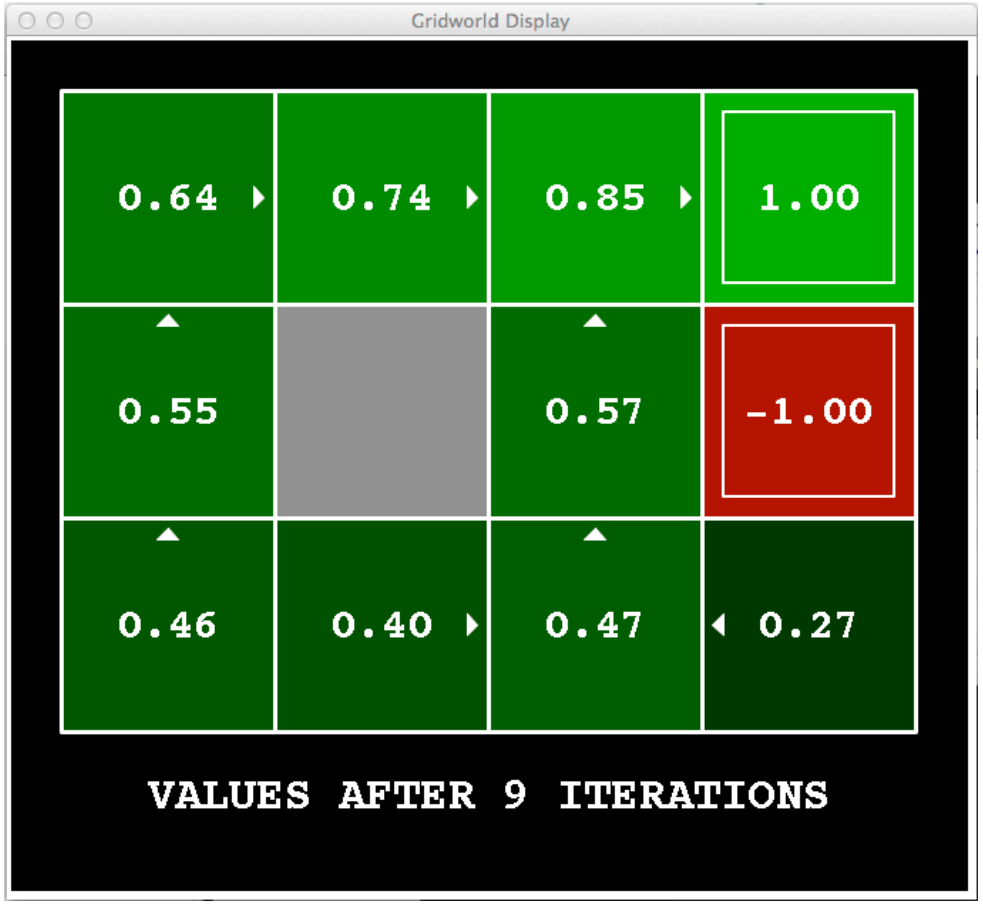
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



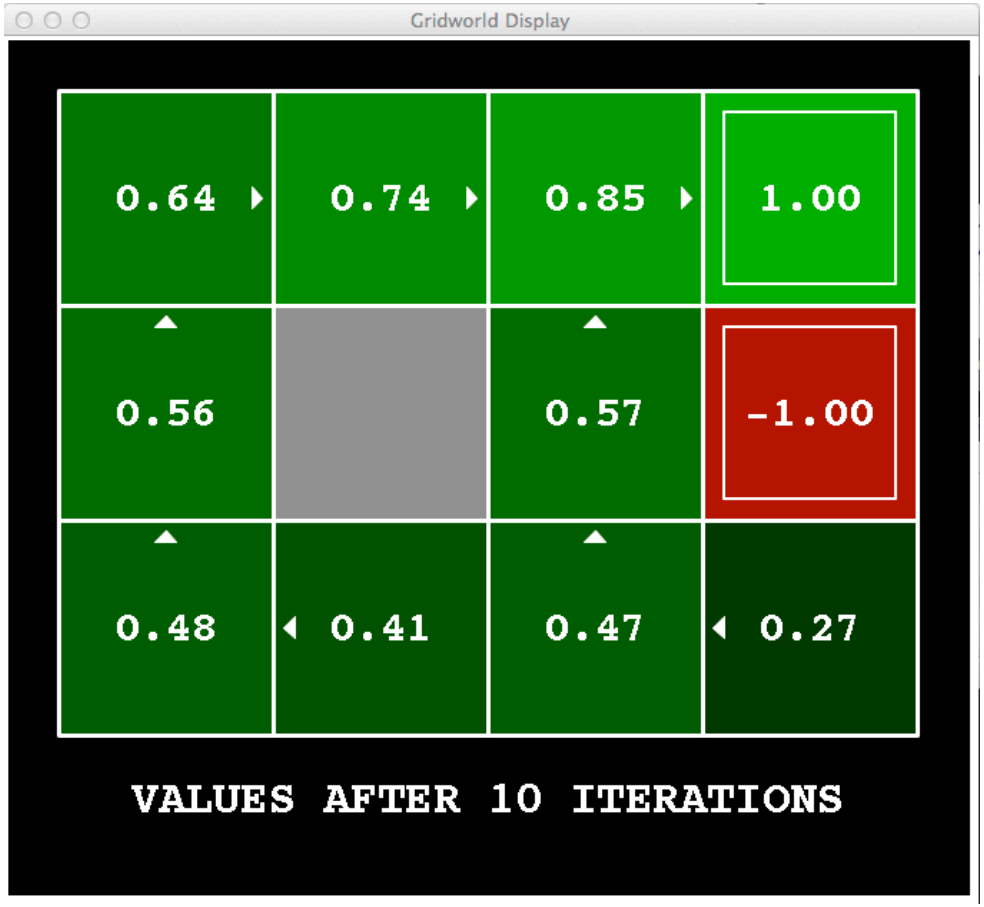
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



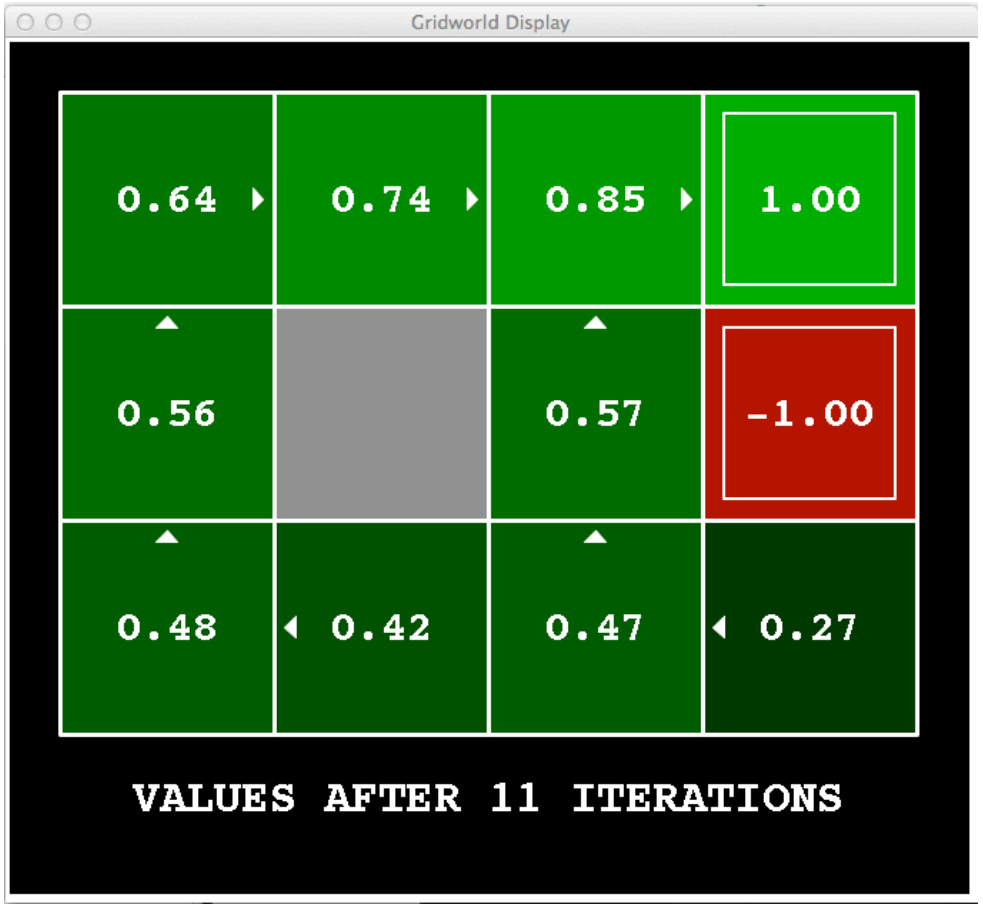
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



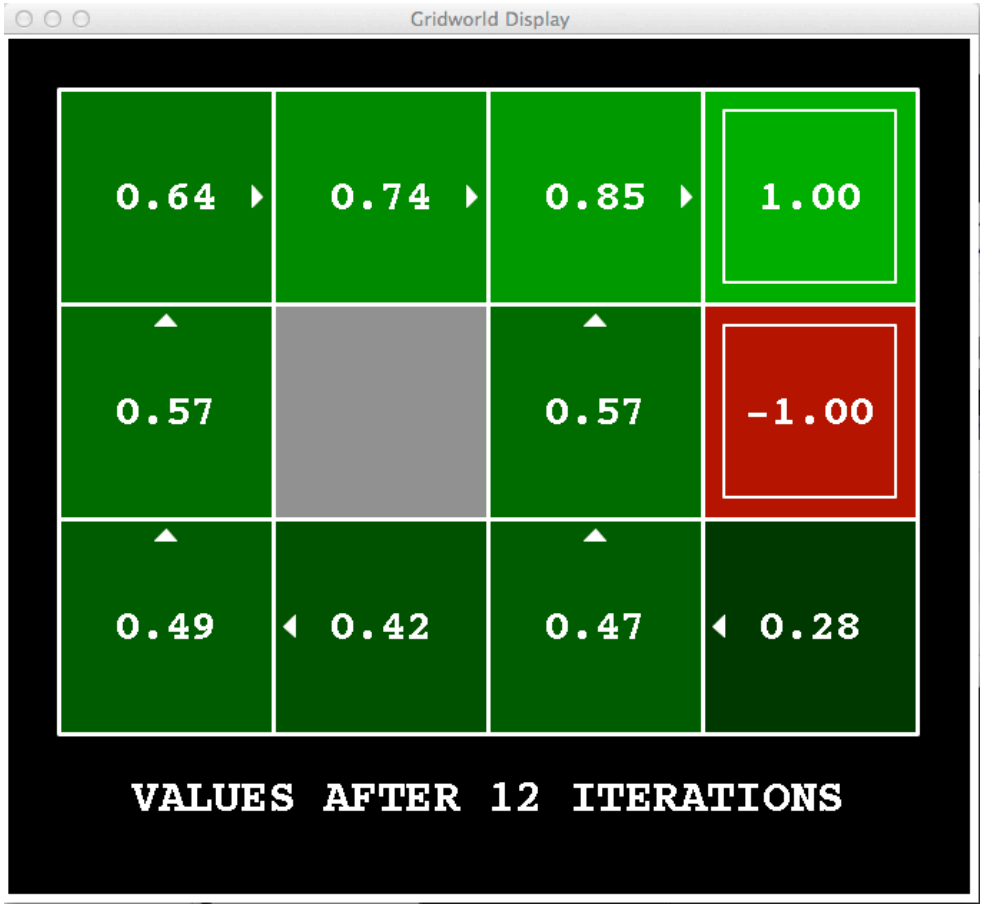
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



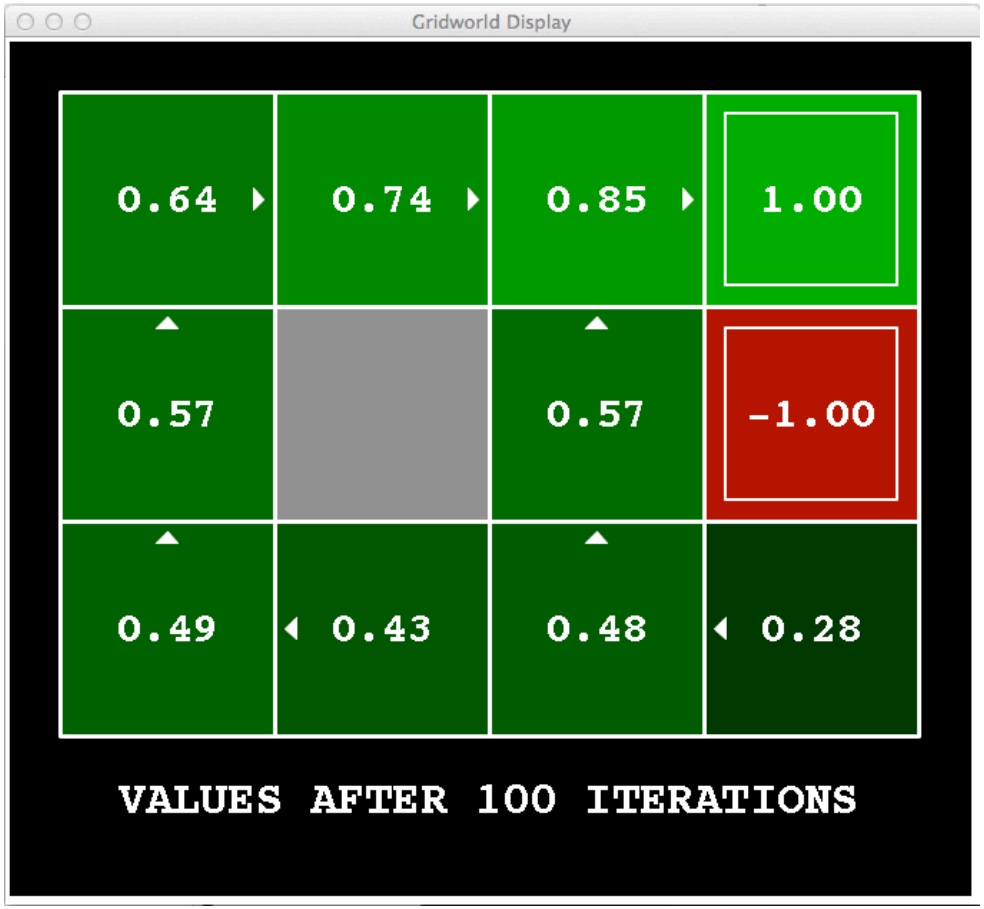
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

策略迭代法 Policy Iteration

- 另一种方法求解最优值:
  - 步骤 1: 策略评价: 给定某个固定的行动策略，计算各个状态值（尽管他们不是代表最优的状态值!），迭代计算，直到这个状态值收敛
  - 步骤 2: 策略改进: 更新行动策略，使用向前一步的计算，使用之前迭代计算收敛的（but not optimal!）状态值（作为未来的状态值）
  - 重复这两步直到行动策略收敛

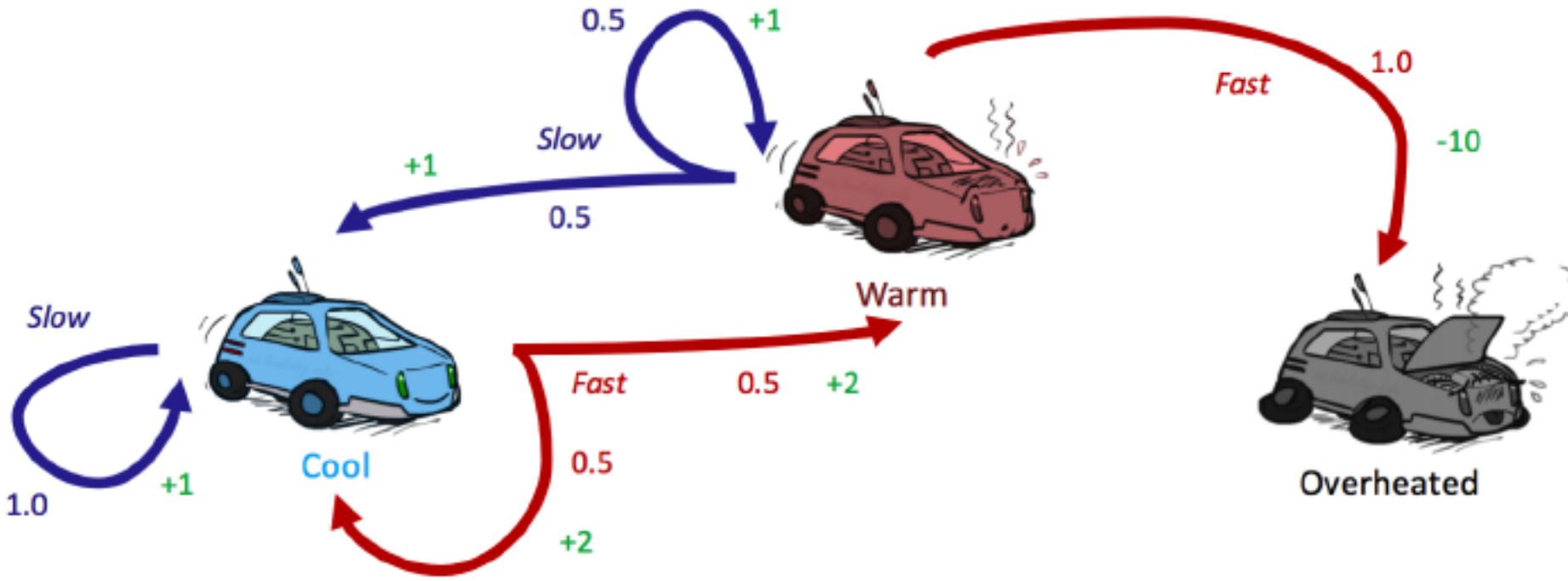
策略迭代法 （Policy iteration）

- 策略评价：对于当前的策略 π，使用策略评价过程计算状态位置的值：
  - 迭代直至V-值收敛

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

- 策略改进：给定计算出来的V-值，通过策略提取过程，计算获得更好一步的策略
  - 向前 一步的 计算：

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



	Cool	Warm	Overheated
π_0	Slow	Slow	
$V^{\pi_0}(s)$			
π_1			

比较

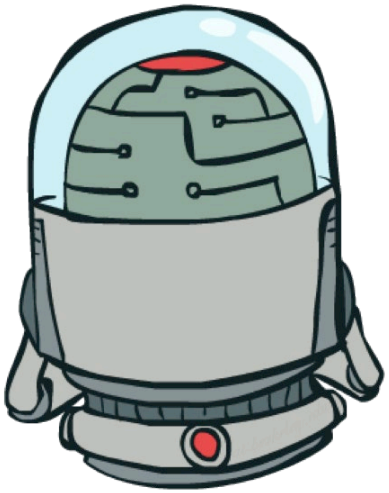
- 状态值迭代和策略迭代方法计算的是同一件事情（所有状态节点的最优功效值）
- 在状态赋值迭代里：
  - 每次迭代，更新状态V-值，和策略(隐式地)
  - 没有直接追踪策略，但是从不同行动分支中获取一个最大值时，实际上隐式地计算了策略
- 在策略迭代里：
  - 我们在当前固定的策略下，通过几次迭代，更新计算状态的功效值(V-值)（每次迭代很快，因为我们此时只考虑一个行动，而不是所有的行动分支）
  - 在当前策略评价过程完成以后，一个新的策略被挑选出来（这一步较慢，就像状态赋值迭代方法里的一次迭代）
  - 新的策略将会更优化（否则的话，迭代过程结束）
- Both are dynamic programs for solving MDPs

总结：MDP 算法

- 我们用到的算法：
  - 计算状态节点的最优功效值：使用*状态赋值迭代*，或*策略迭代*方法
  - 给定一个策略，计算状态值：使用*策略评价*方法
  - 通过状态值获取一个策略：使用策略提取方法（向前一步优化）

- 这些方法看上去都很像！
  - 它们本质上都是基于 Bellman 赋值更新表达式
  - 全都使用了基于期望最大值的向前一步优化计算模块
  - 它们区别只是在于是否固定一个策略，或是在所有行动分支中进行最优挑选（最大化期望功效值的行动）

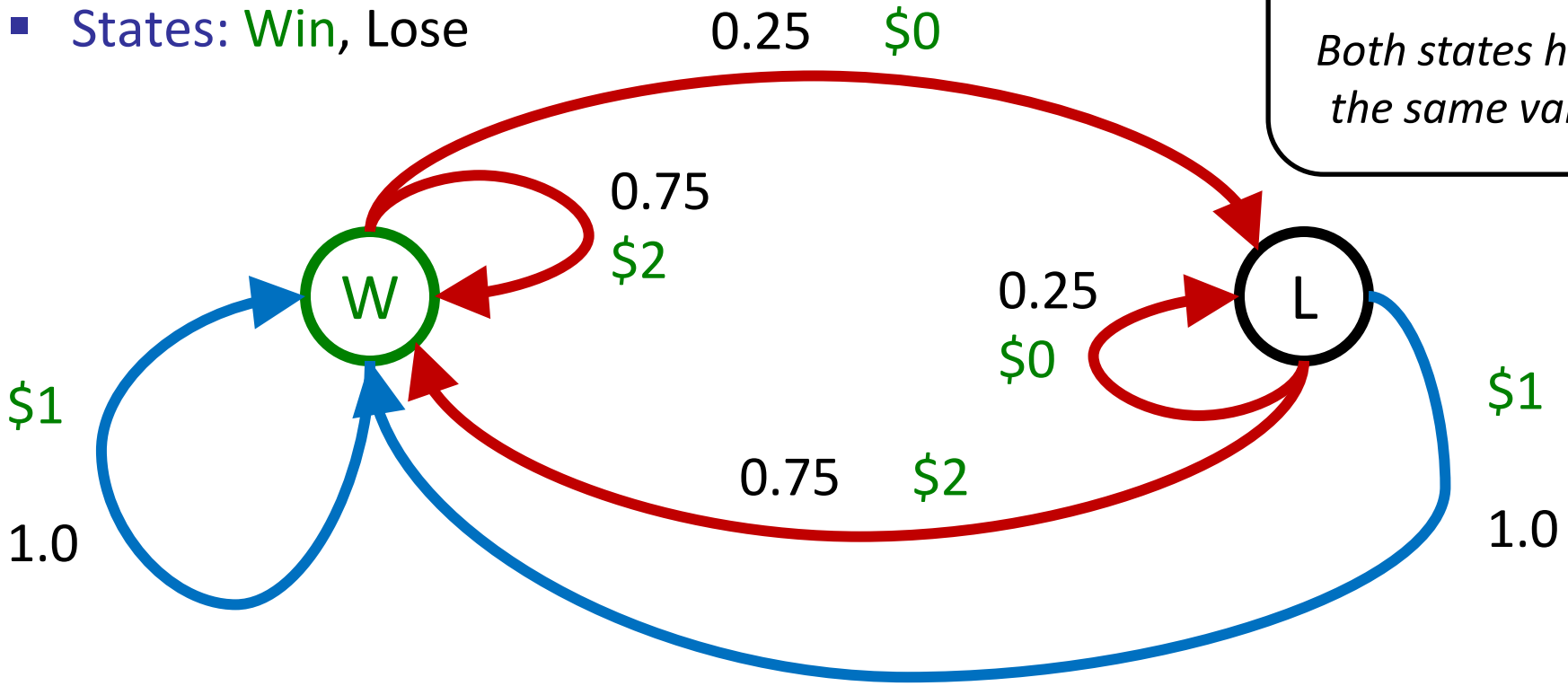
Double Bandits



Double-Bandit MDP

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

No discount
100 time steps
Both states have the same value



Offline Planning

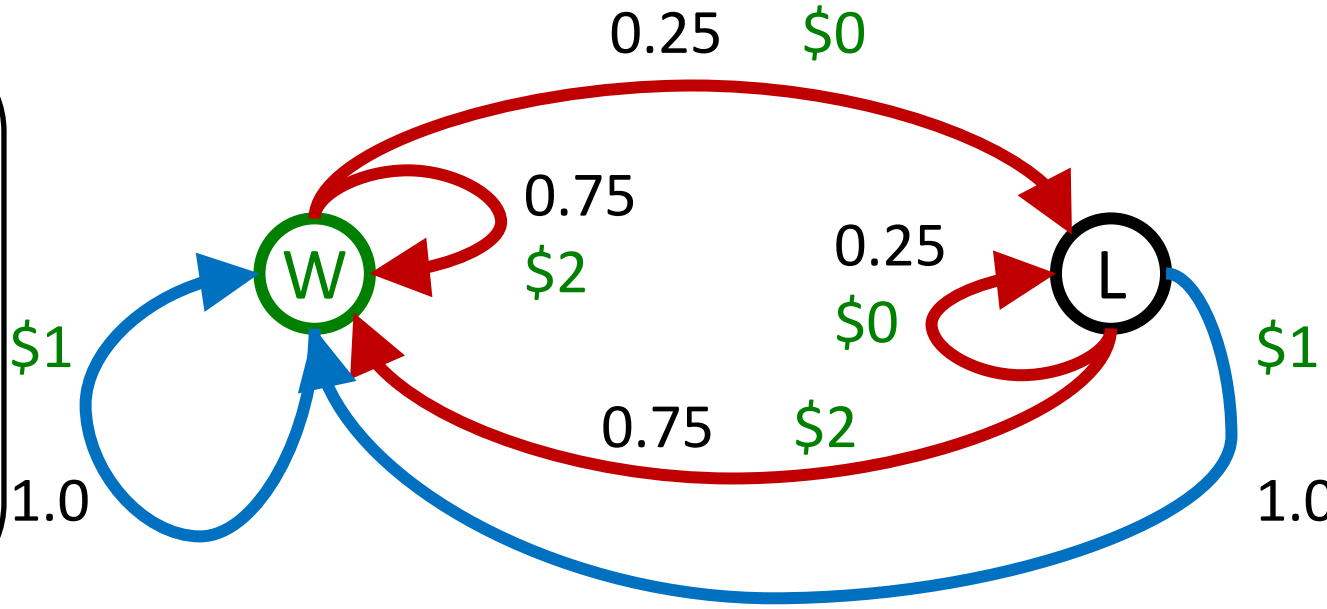
- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

No discount

100 time steps

Both states have the same value

	Value
Play Red	150
Play Blue	100



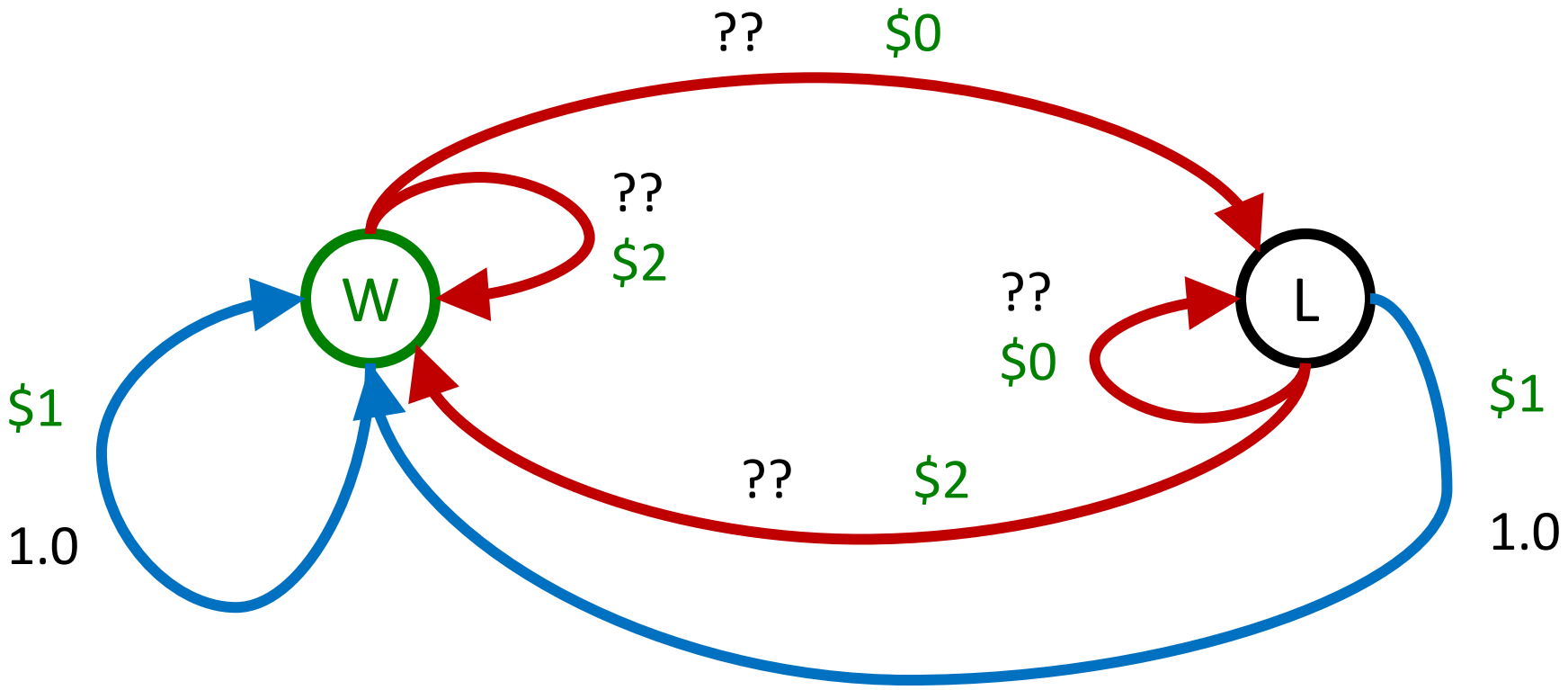
Let's Play!

$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 (增强学习)
  - 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 （探索）: you have to try unknown actions to get information
  - Exploitation （利用）: 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!