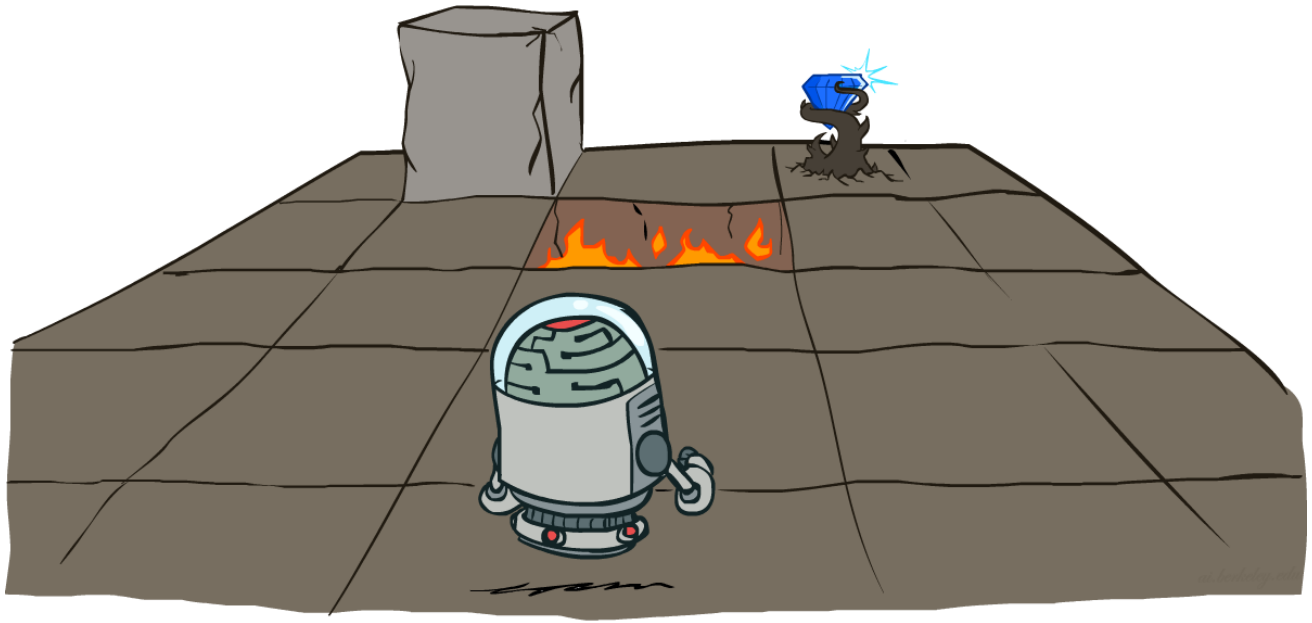
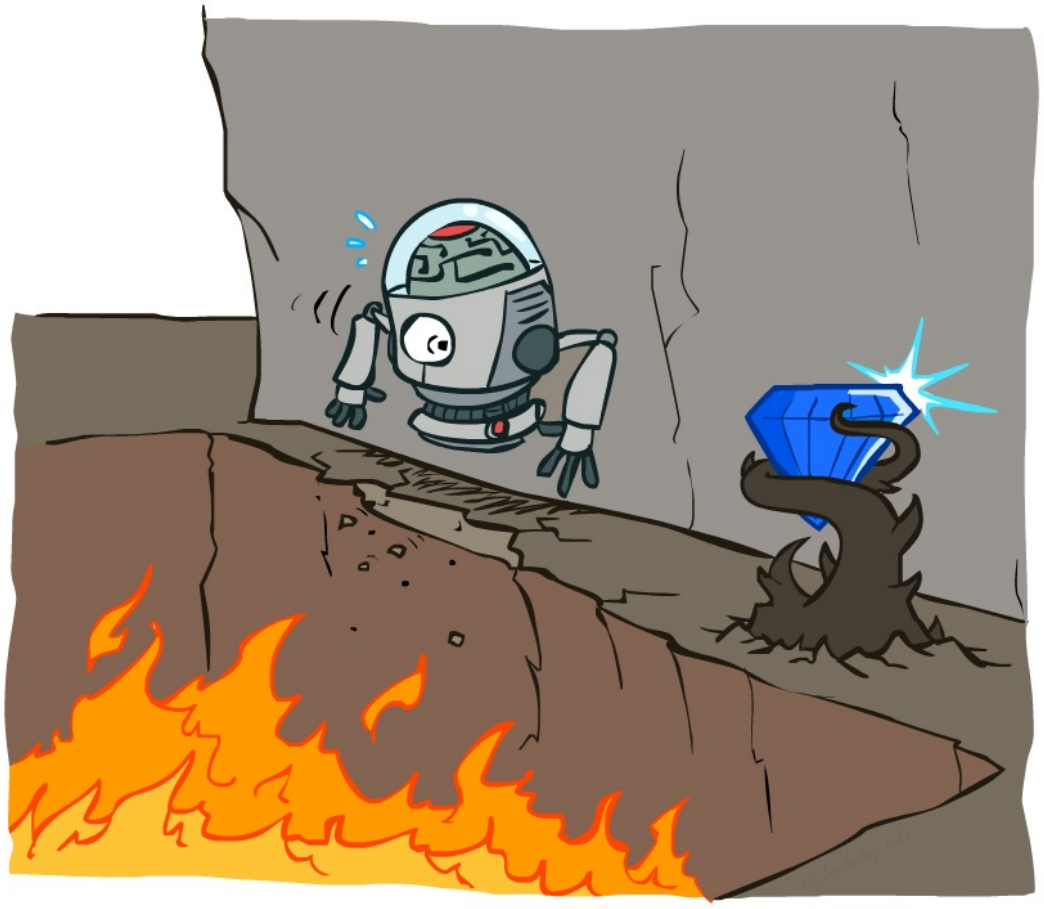
# Markov Decision Processes



AIMA Chapter 17

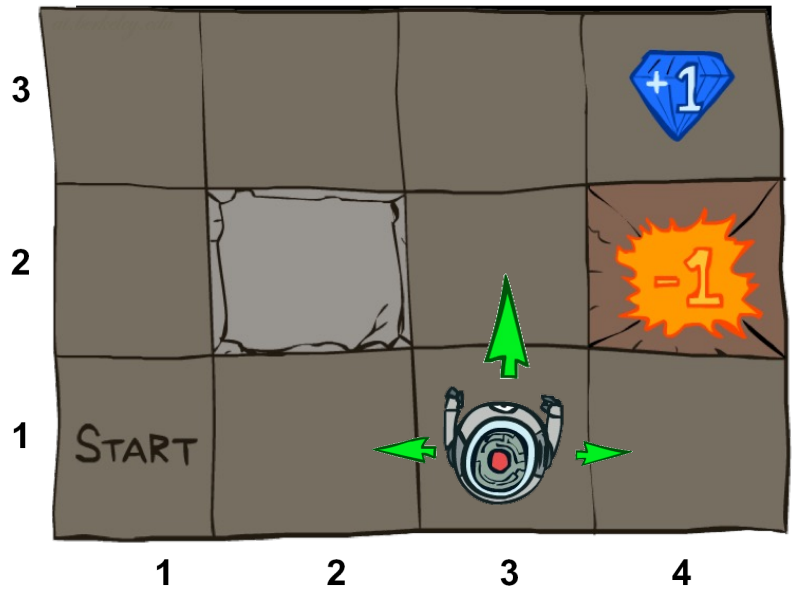
[Adapted from slides by Dan Klein and Pieter Abbeel at UC Berkeley]

# Non-Deterministic Search



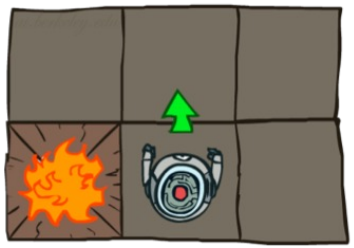
# Example: Grid World

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

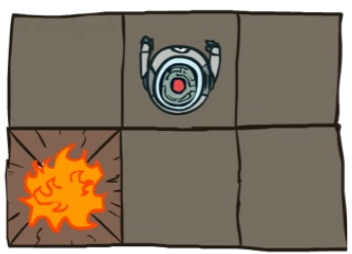


# Grid World Actions

## Deterministic Grid World



搜索问题的解就是一个 plan，提前就可以得到.



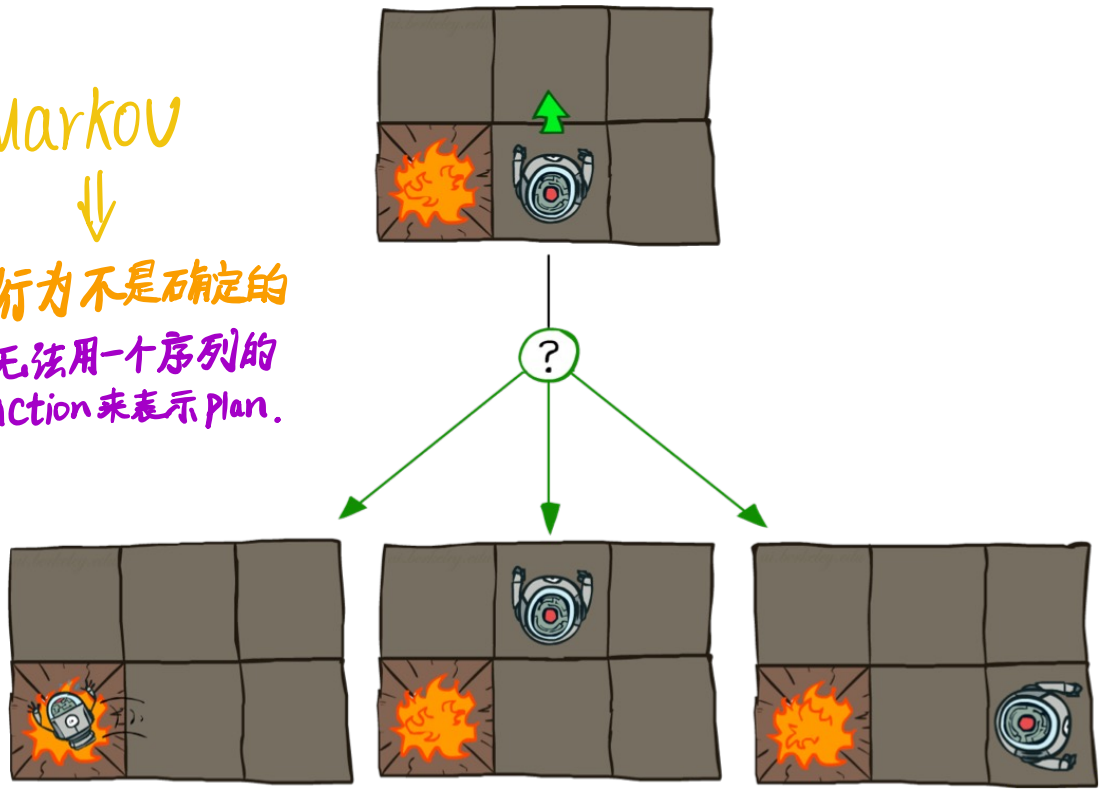
## Stochastic Grid World

Markov

⇓

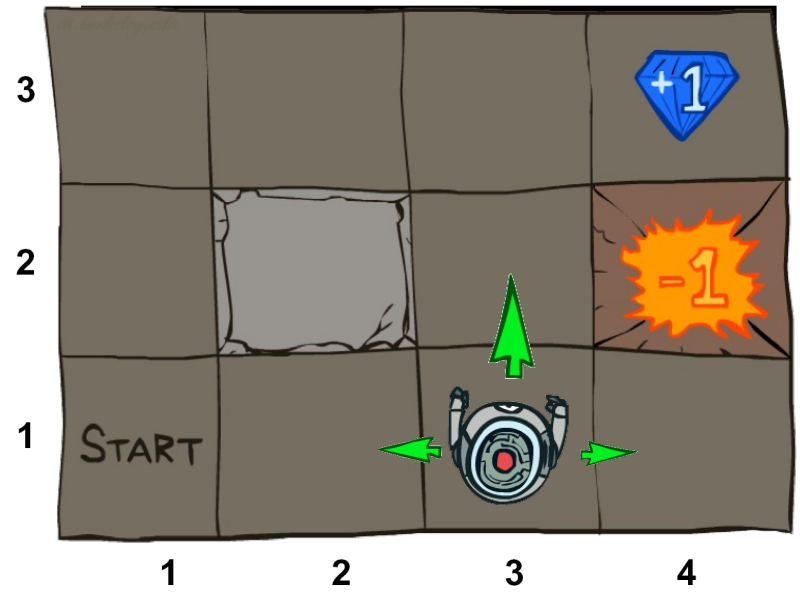
行为不是确定的

无法用一个序列的 action 来表示 plan.



# Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$ (N,W,S,E)∈A
  - A transition function $T(s, a, s')$ in Markov, transition is P.
    - 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 e.g. 步长(走到n步, stop)




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

# What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent (通常对概率做了某些假设)

- 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)

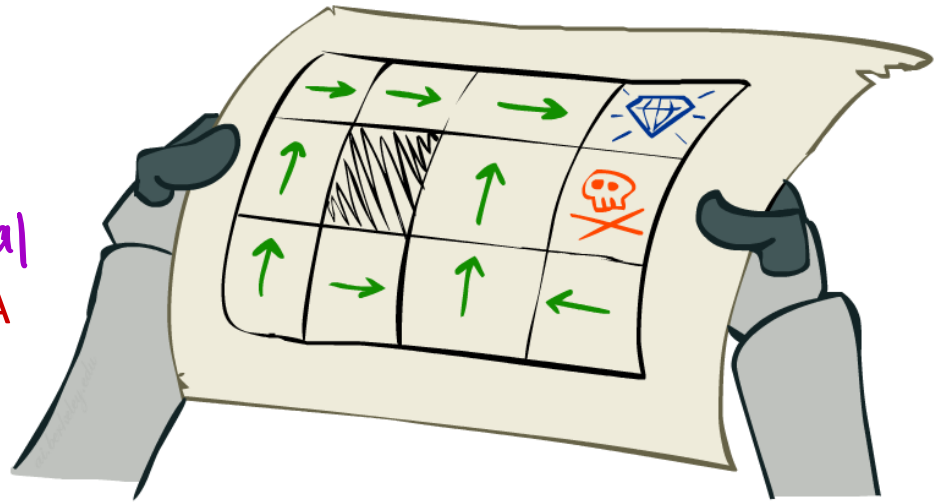
# Policies

- 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 $\pi$ gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

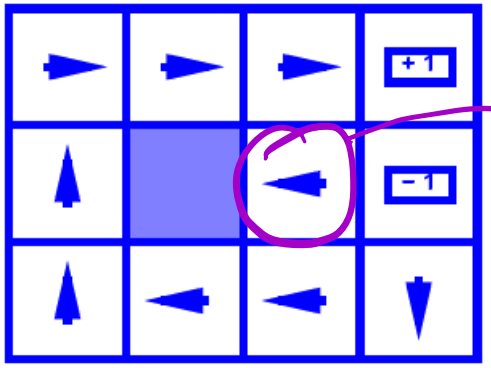
optimal

不考虑该行为的未来的后果.



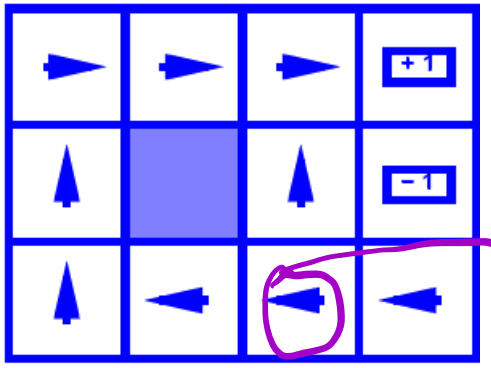
# Optimal Policies

R(s) = “living reward”




R(s) = -0.01

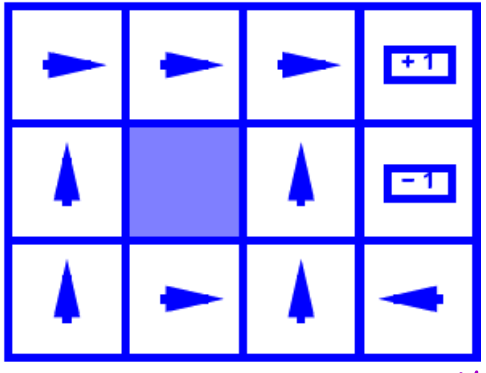
80% ← : stop
10% ↑
10% ↓
∴ 永远不会往右



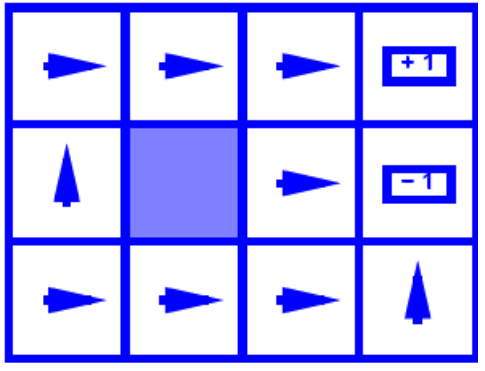

R(s) = -0.03

choose 更远

cost变高了，尽快结束游戏

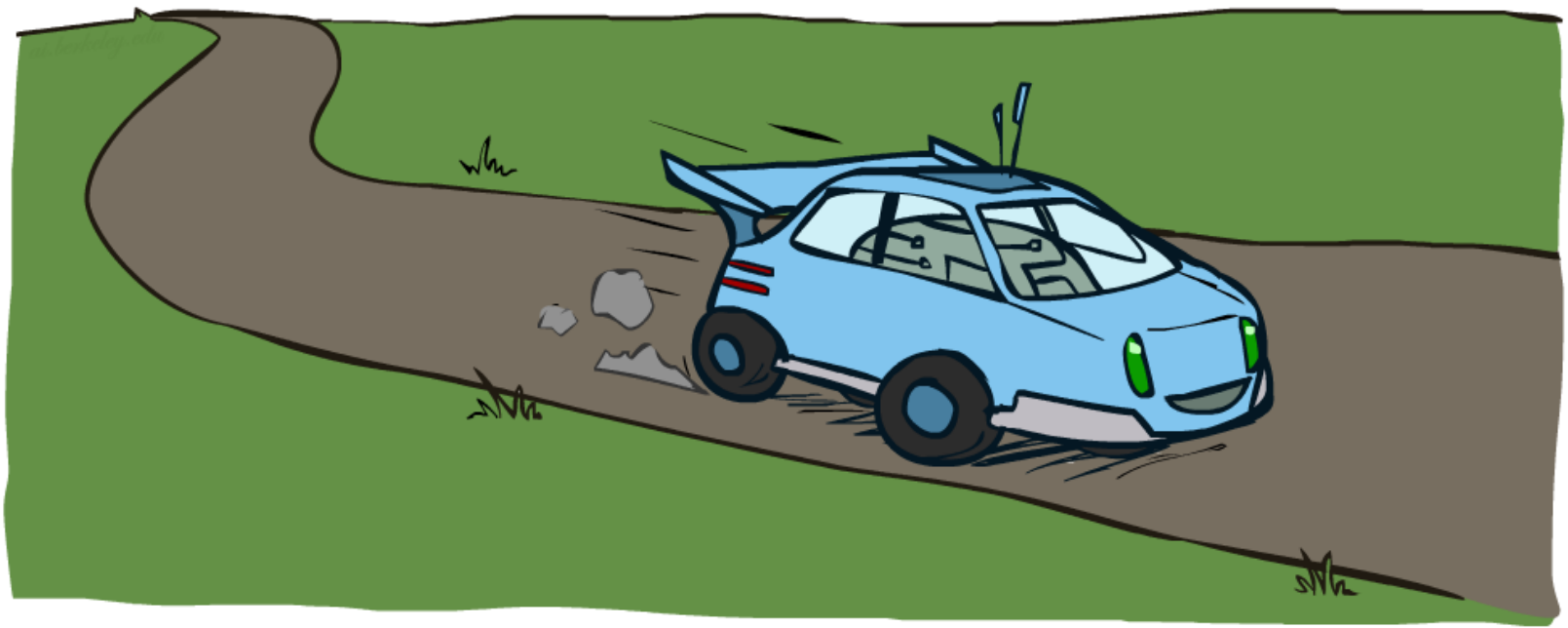



R(s) = -0.4 倾向于找更短path



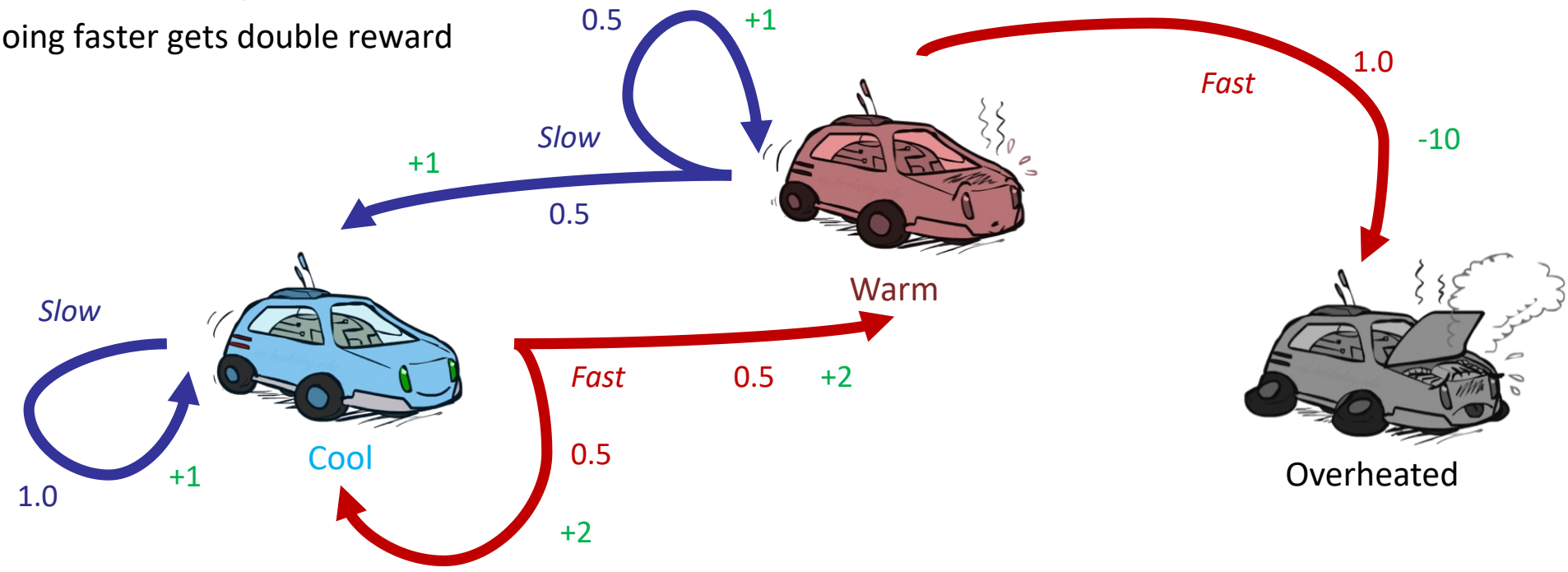

R(s) = -2.0 尽快结束游戏.

# Example: Racing

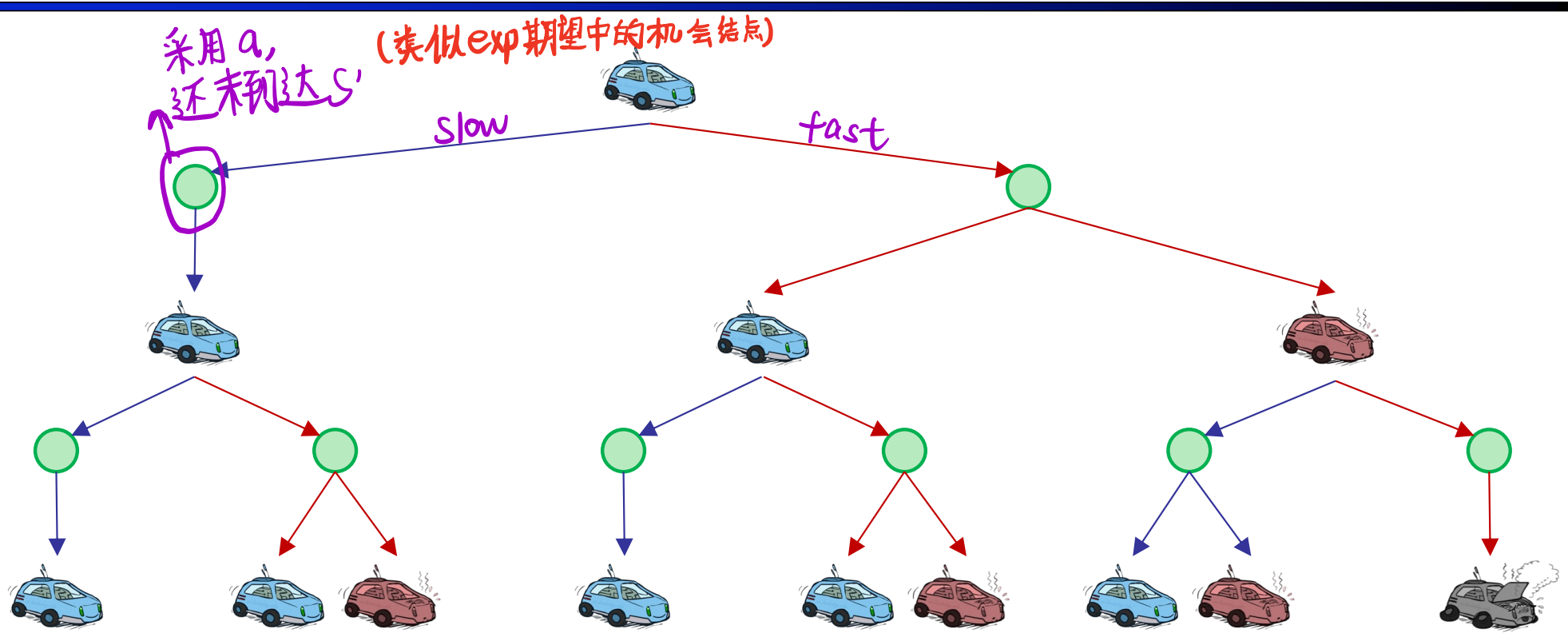


# Example: Racing

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

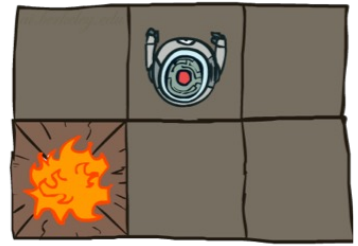


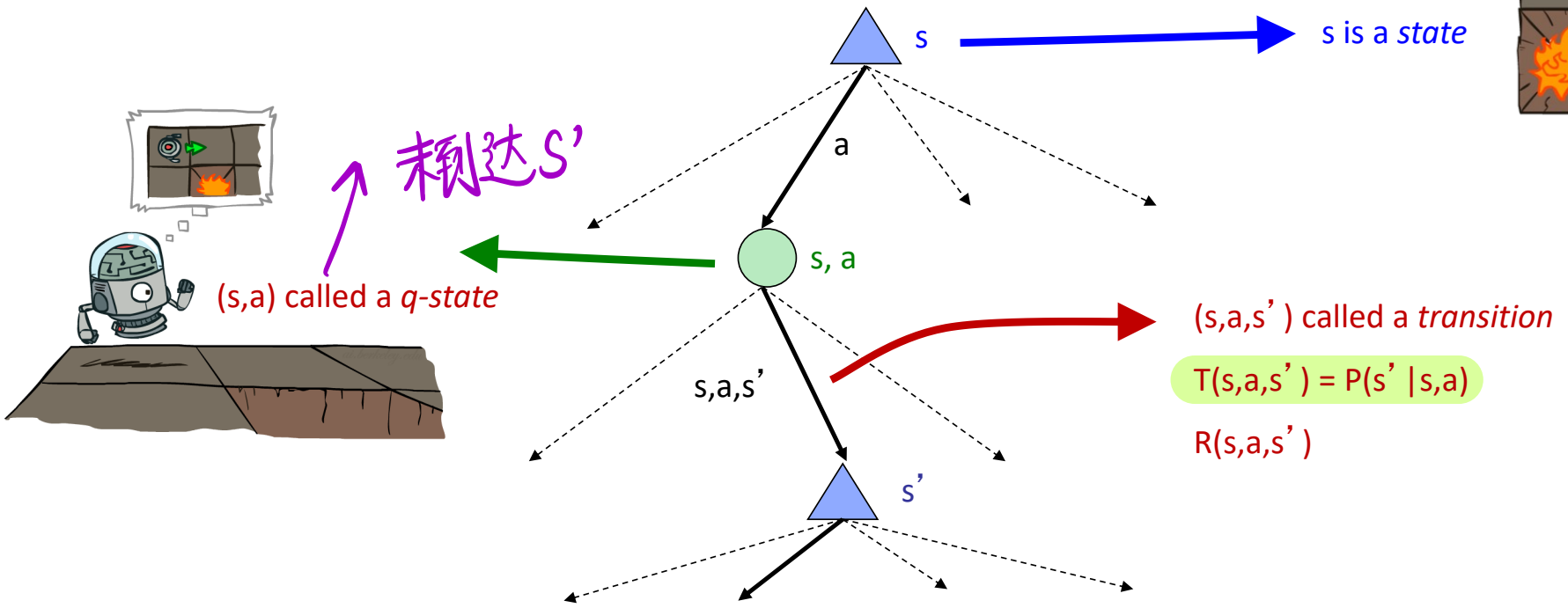
# Racing Search Tree



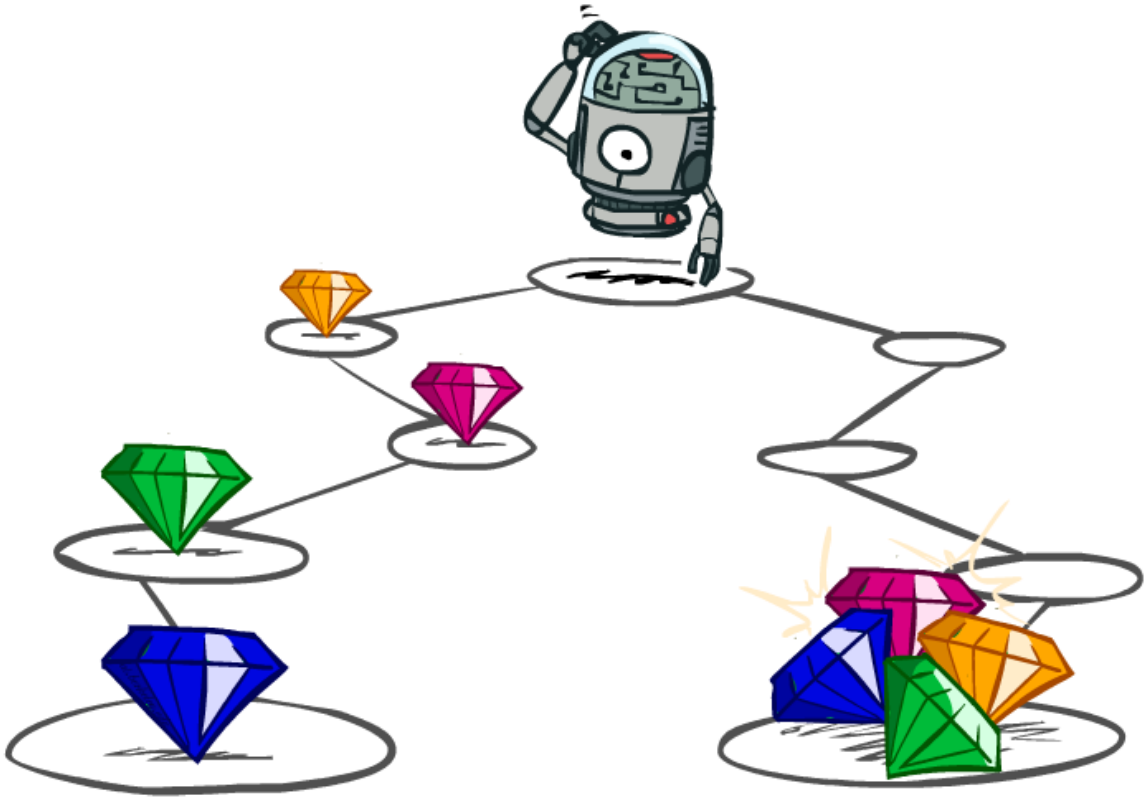
# MDP Search Trees

- Each MDP state projects an expectimax-like search tree





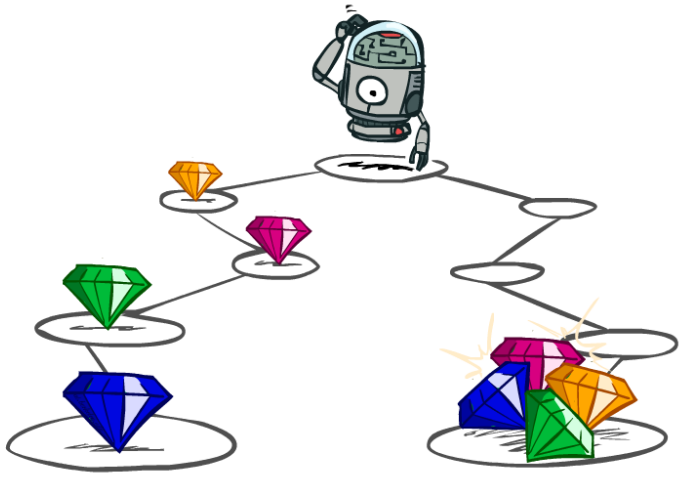
# Utilities of Sequences



更及时获得更多的奖励.

# Utilities of Sequences

- What preferences should an agent have over reward sequences?
- More or less? [1, 2, 2] or [2, 3, 4]
- Now or later? [0, 0, 1] or [1, 0, 0]



# 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

$\gamma$

Worth Next Step

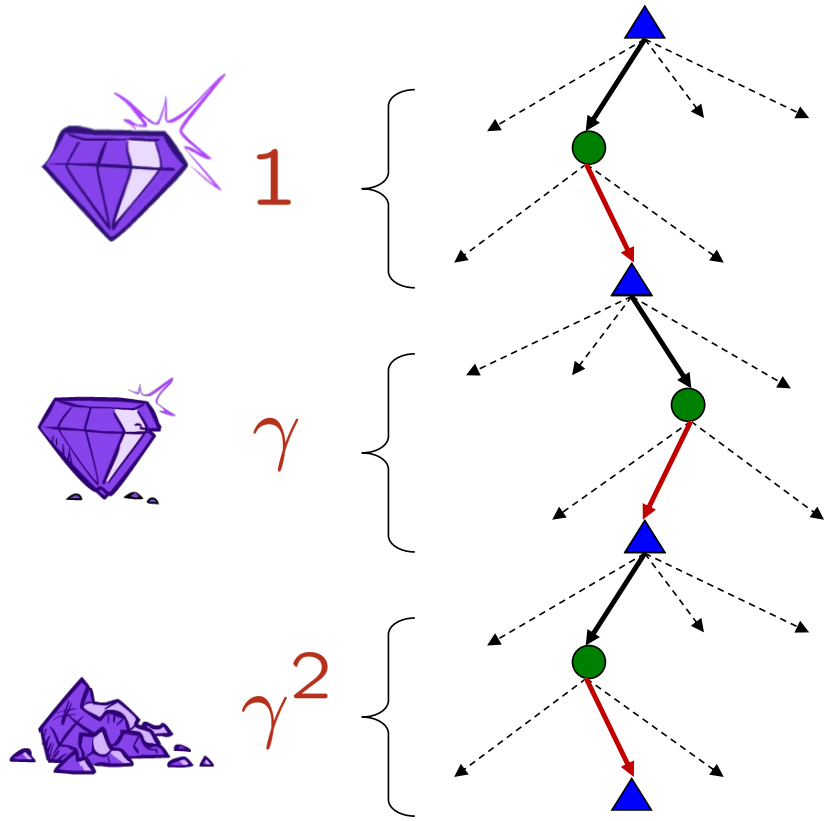
$\gamma^2$

Worth In Two Steps

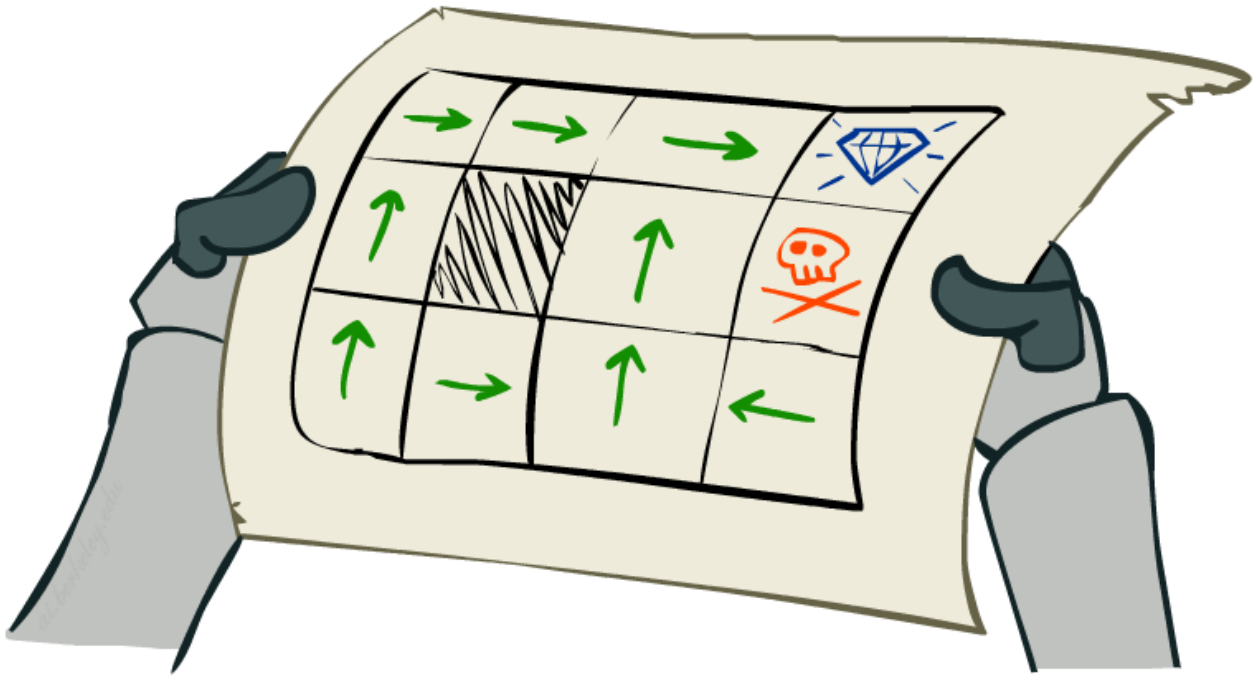
# Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])

帮助算法快速收敛.



# Solving MDPs



# Optimal Quantities

从s出发一直采用最优(累积optimal)

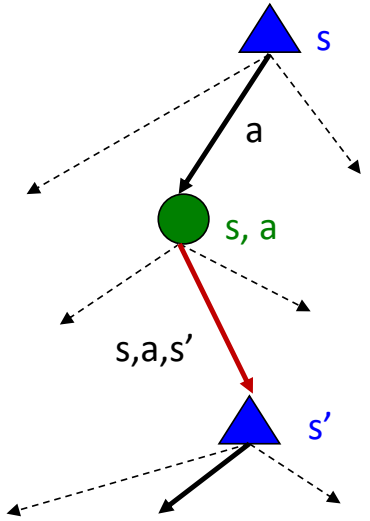
- The value (utility) of a state s:
  $V^*(s)$ = expected utility starting in s and acting optimally

Q状态开始，后续最优

- 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

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

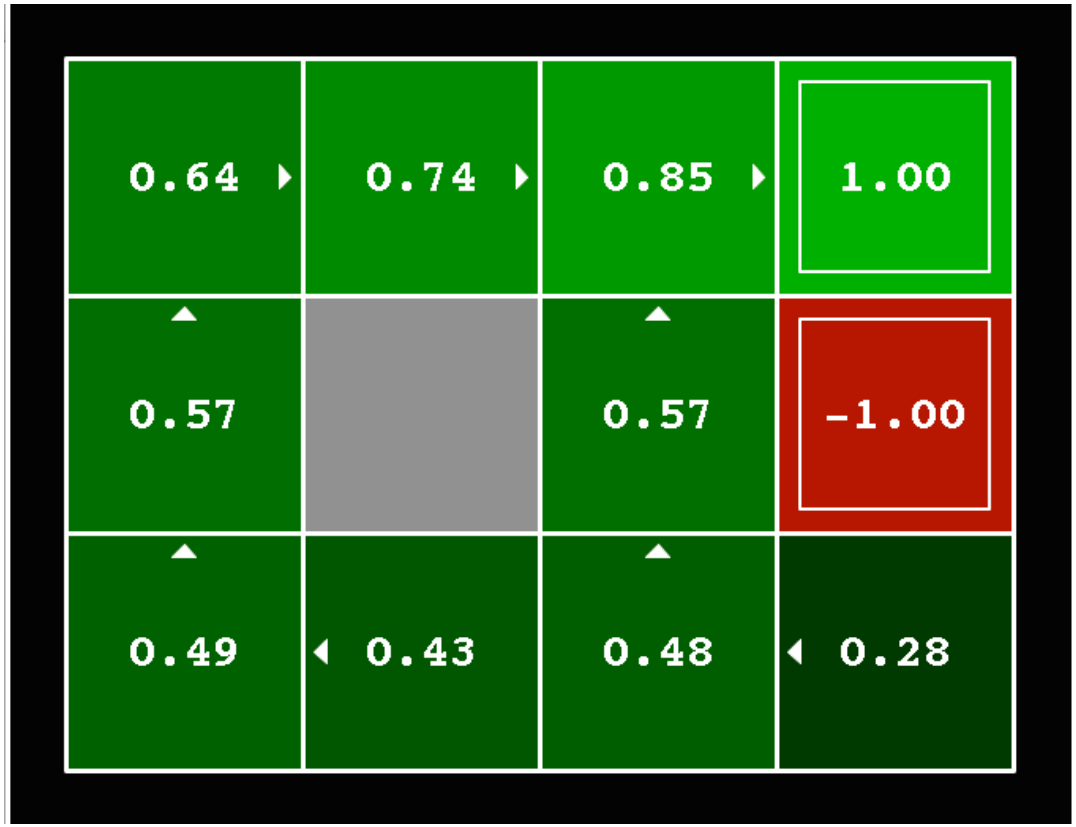



s is a *state*

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

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

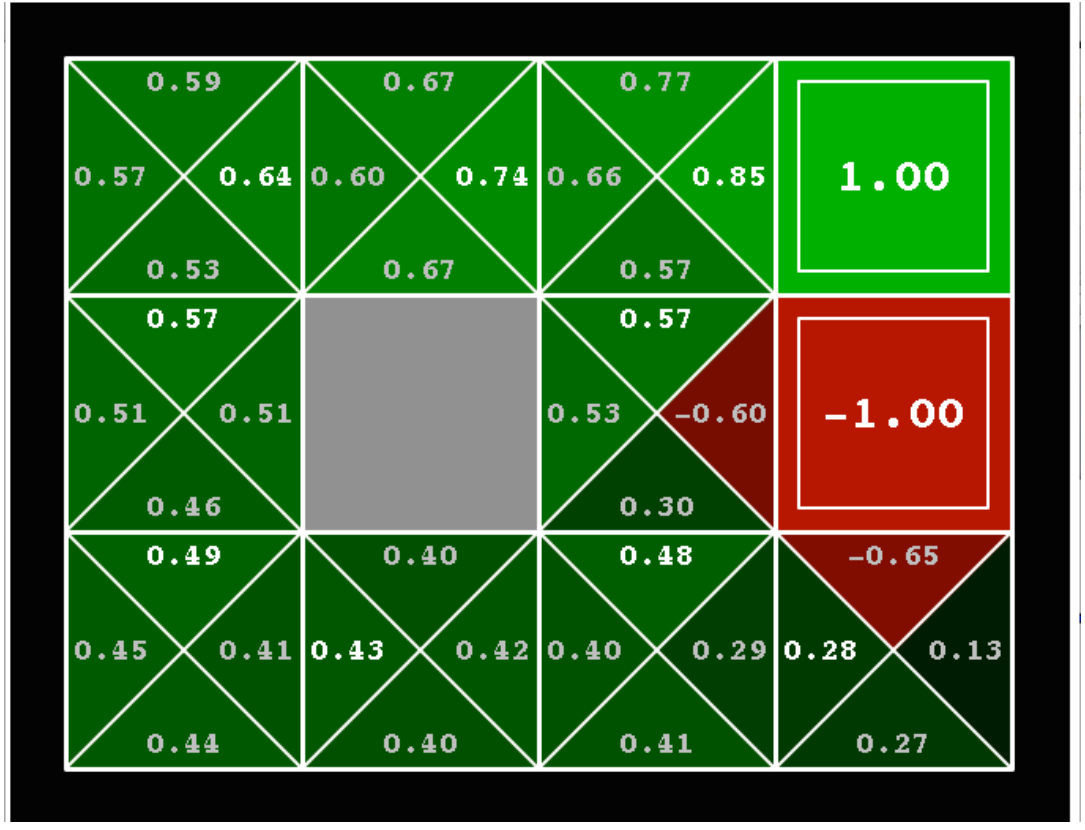
# Gridworld V Values




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

Noise = 0.2
Discount = 0.9
Living reward = 0

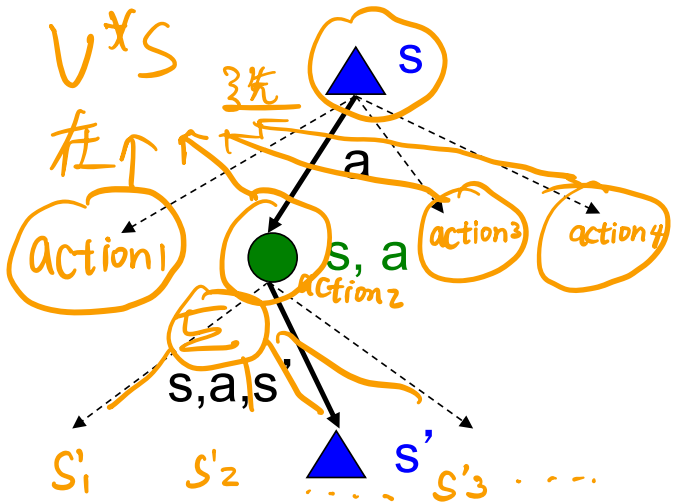
# Gridworld Q Values




Noise = 0.2
Discount = 0.9
Living reward = 0

# Values of States

- How to compute the value of a state
  - Expected utility under optimal action
  - This is just what expectimax computed!




- Recursive definition of value:

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

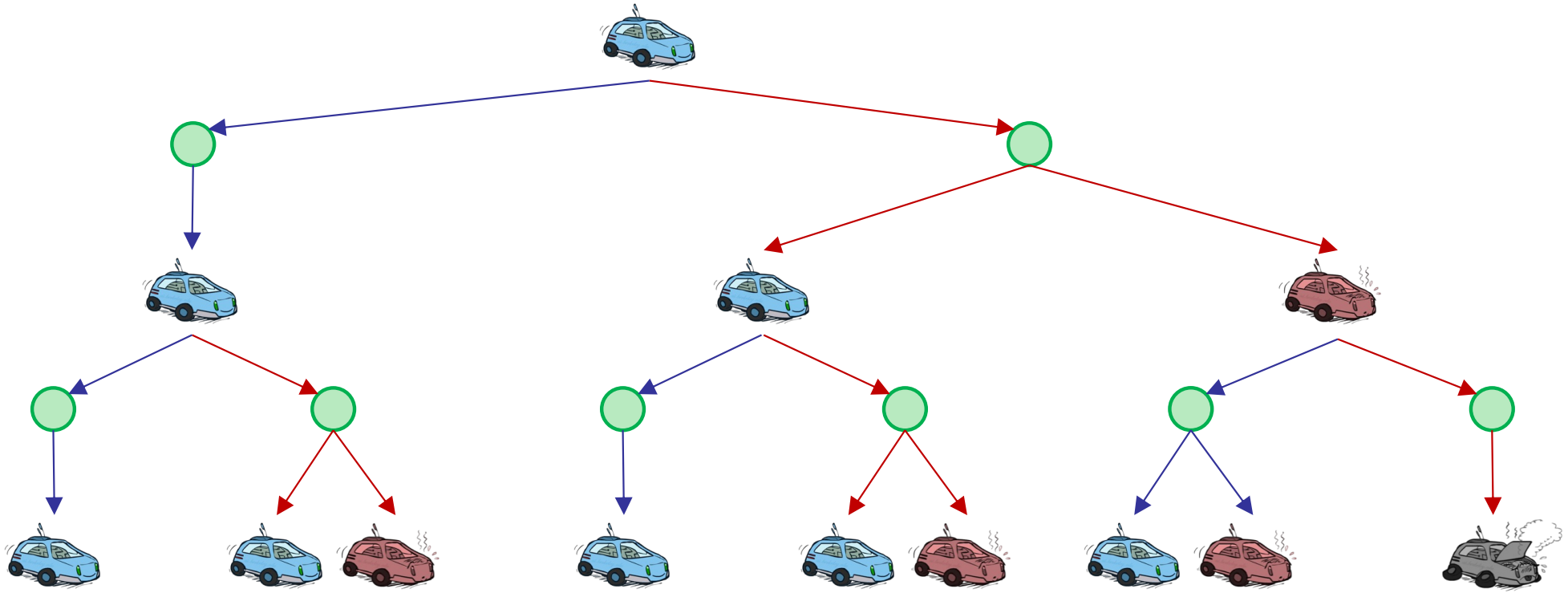
$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma V^*(s') \right]$$

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

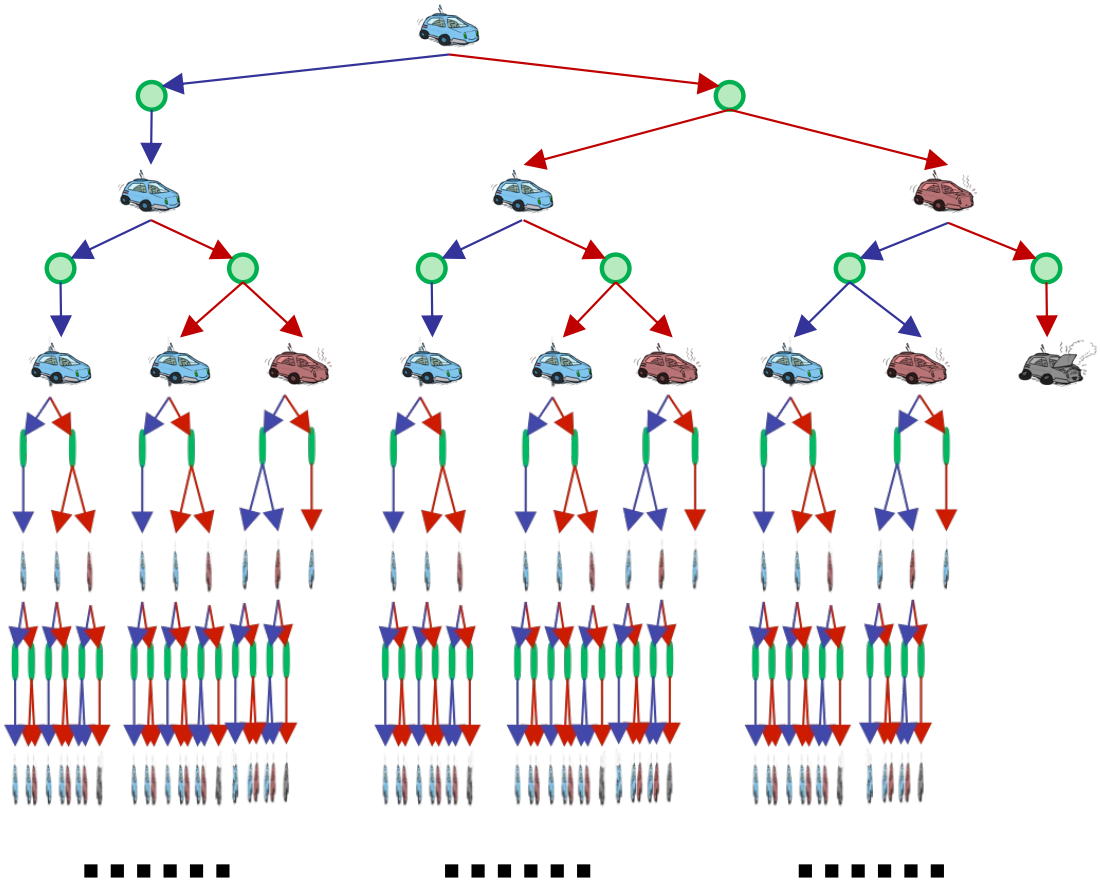
$P(s'|s,a)$

The Bellman Equation

# Racing Search Tree



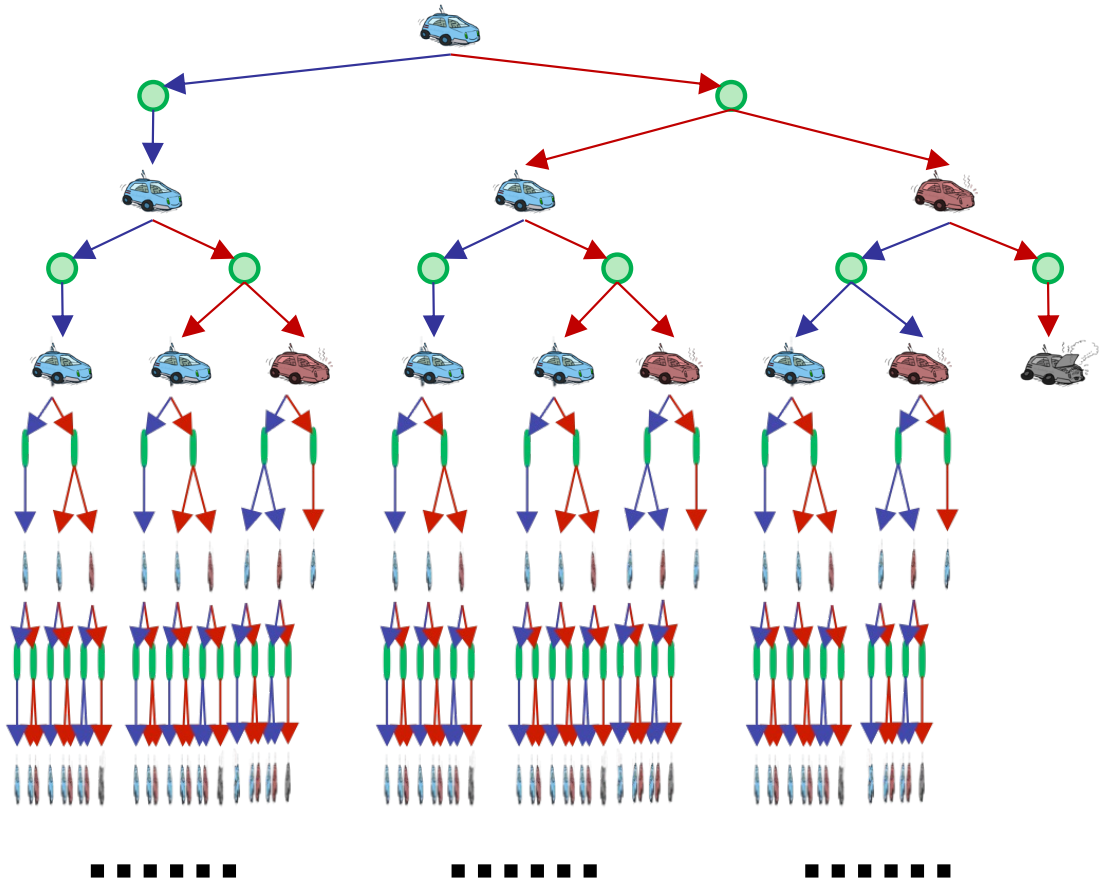
# Racing Search Tree



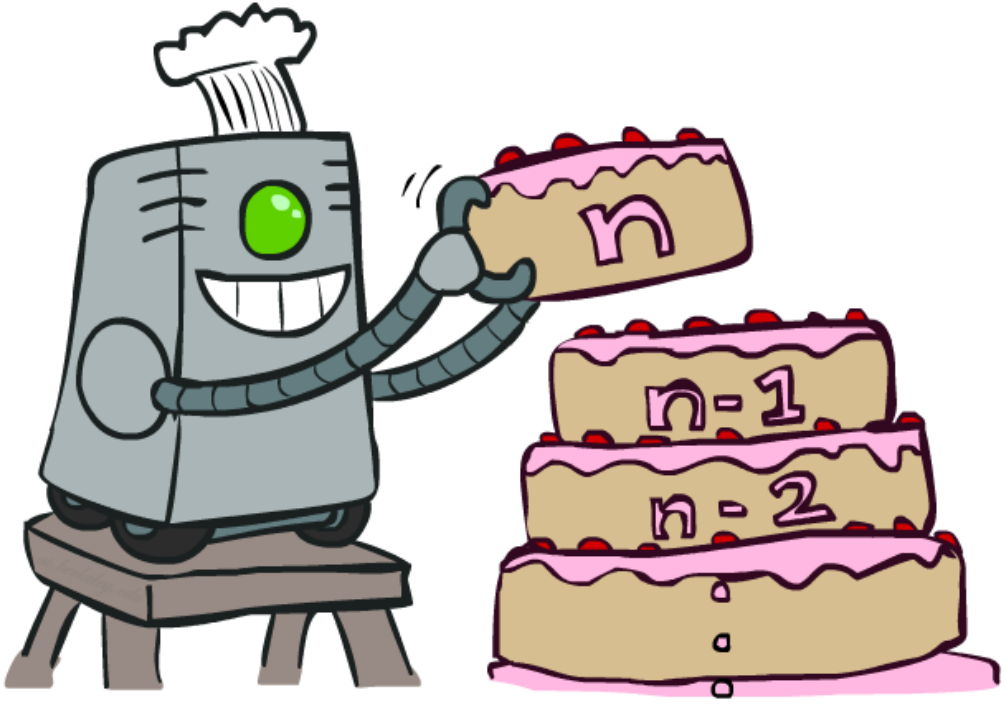
# Problems with Expectimax

- Problem 1: States are repeated
  - Idea: Only compute needed quantities once
- Problem 2: 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 $\gamma < 1$

γ特别小的时候不影响了.

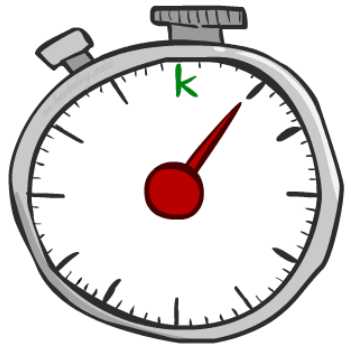


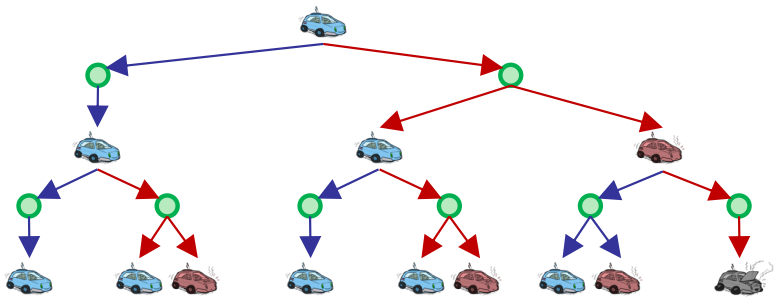
# Value Iteration

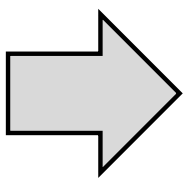


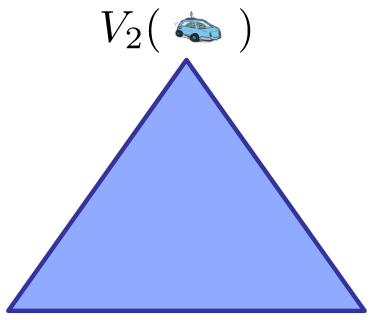
# Time-Limited Values

- 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

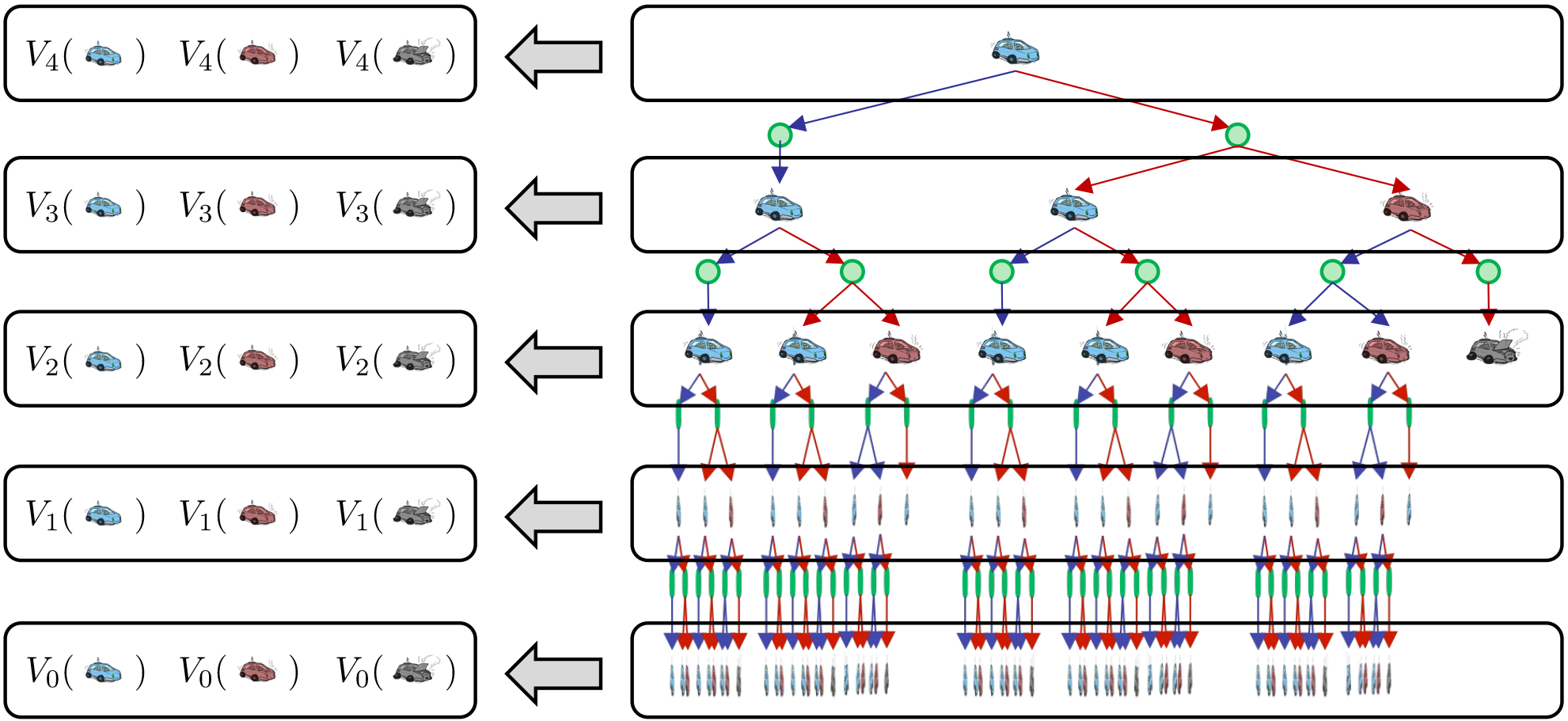






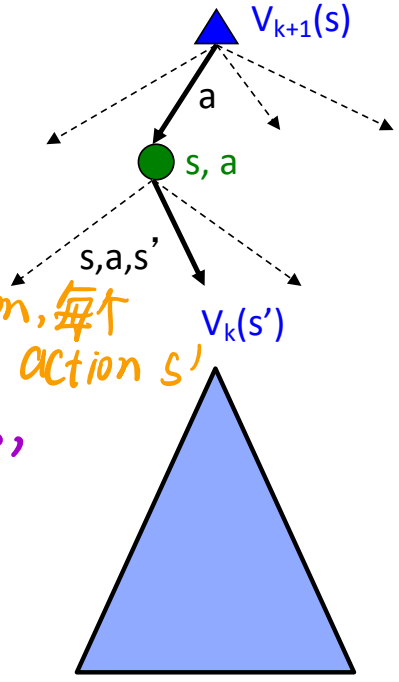
# Computing Time-Limited Values



# 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:

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




在k+1时刻有 S个状态，每个状态有A的action，每个action s'

k+1有状态S，k有状态S，

action有 A

∴ $A \cdot S^2$

- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values

唯一最大的值

# Example: Value Iteration

$V_2$

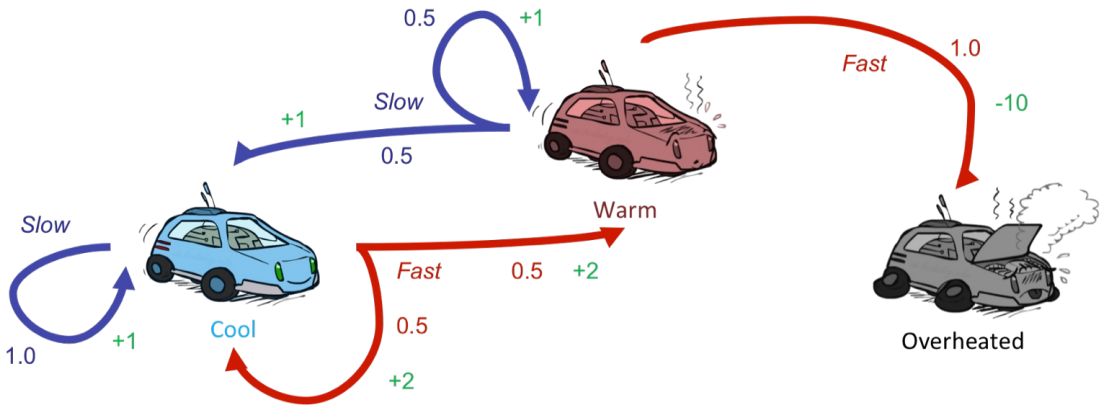
$V_1$

S: 1

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

$V_0$

0 0 0




*Assume no discount!*

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

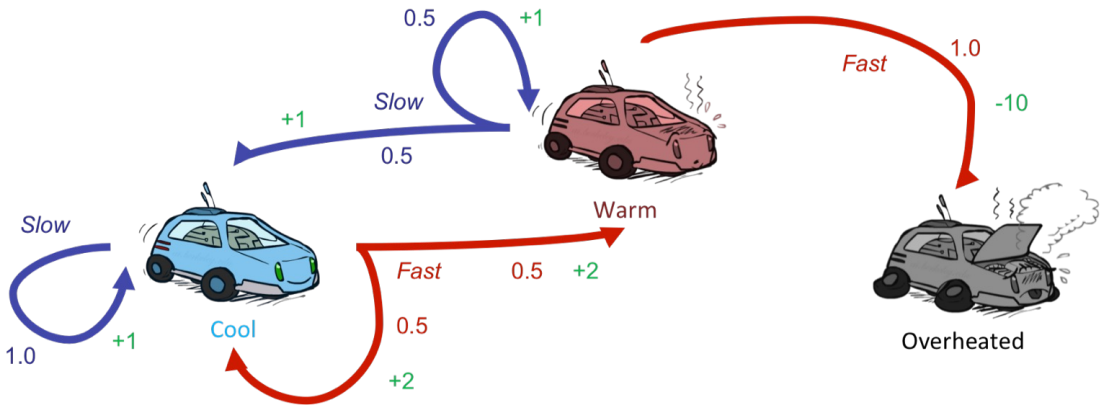
$Q_1(\text{Cool}, S) = \sum_{s'} P(s' \mid \text{Cool}, \text{Slow}) [R(s', \text{Cool}, \text{Slow}) + \gamma V_0(s')]$

# Example: Value Iteration

$V_2$

$V_1$ | 2 | S: .5*1+.5*1=1<br>F: -10

$V_0$ | 0 | 0 | 0




*Assume no discount!*

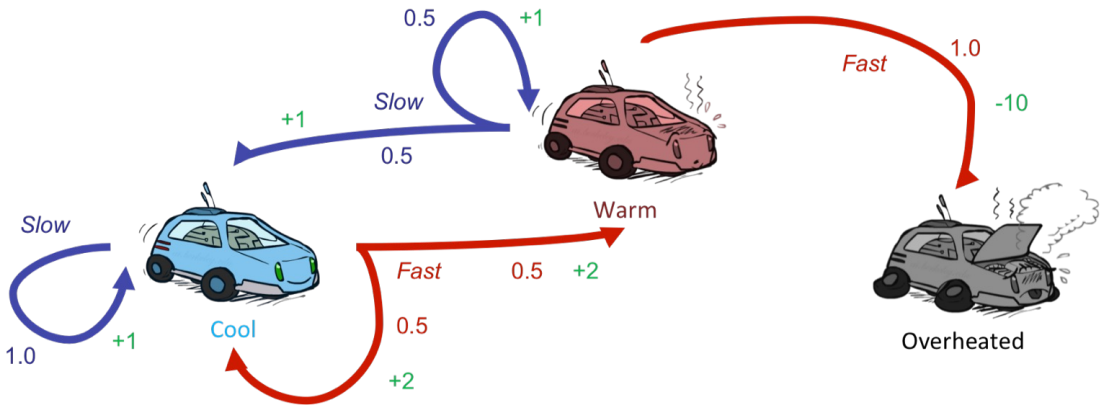
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma V_k(s') \right]$$

# Example: Value Iteration

$V_2$

$V_1$ | 2 | 1 | 0

$V_0$ | 0 | 0 | 0




*Assume no discount!*

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

# Example: Value Iteration

$V_2$

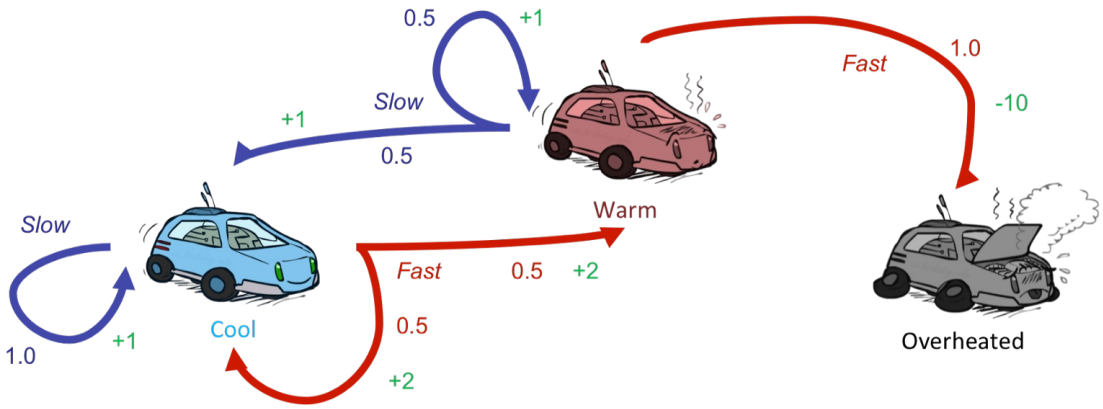
S: 1+2=3
F: .5*(2+2)+.5*(2+1)=3.5

$V_1$

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

$V_0$

| 0 | 0 | 0 |
|---|---|---|




*Assume no discount!*

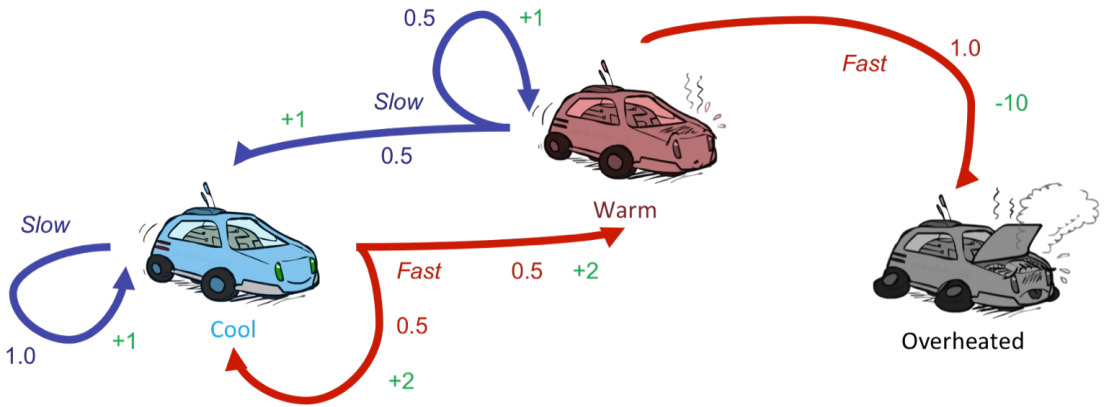
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma V_k(s') \right]$$

# Example: Value Iteration

$V_2$ | 3.5 | S: .5*(2+1)+.5*(1+1)=2.5<br>F: -10

$V_1$ | 2 | 1 | 0

$V_0$ | 0 | 0 | 0

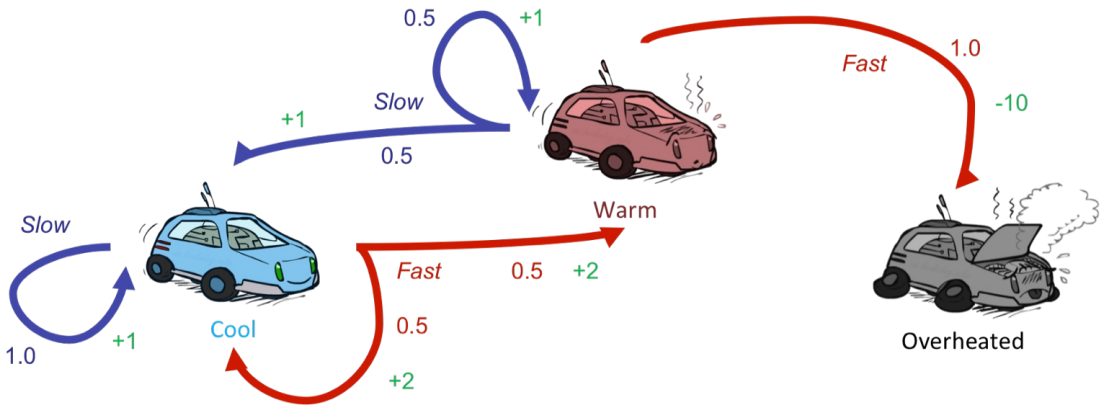



*Assume no discount!*

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

# Example: Value Iteration

| | | | |
|---|---|---|---|
| $V_2$ | 3.5 | 2.5 | 0 |
| $V_1$ | 2 | 1 | 0 |
| $V_0$ | 0 | 0 | 0 |

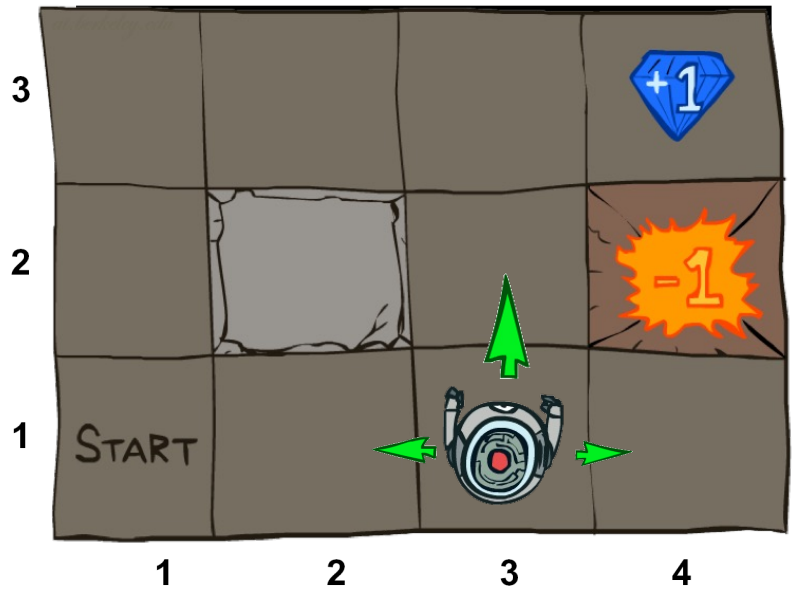



*Assume no discount!*

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

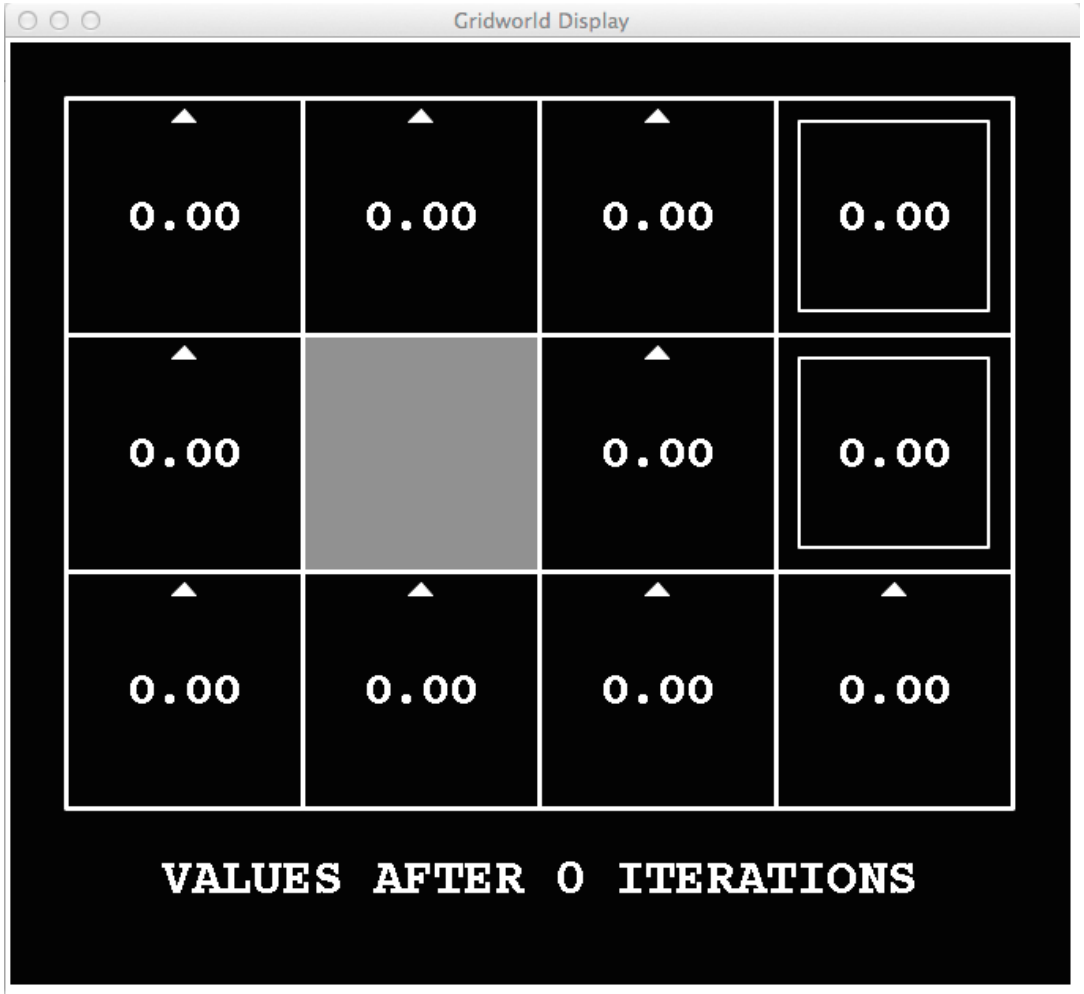
# Example: Grid World

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



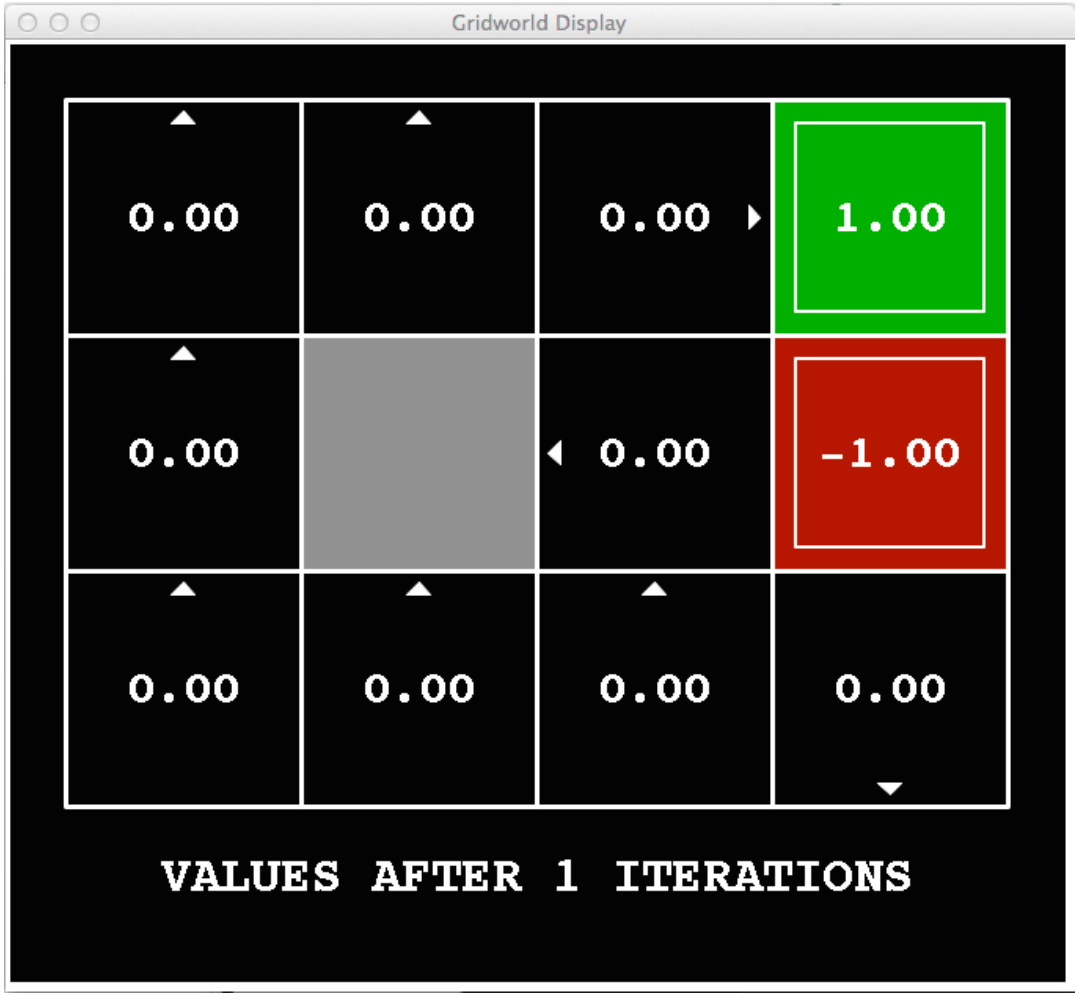

Suppose we get this reward by taking an "exit" action at a goal state

# k=0



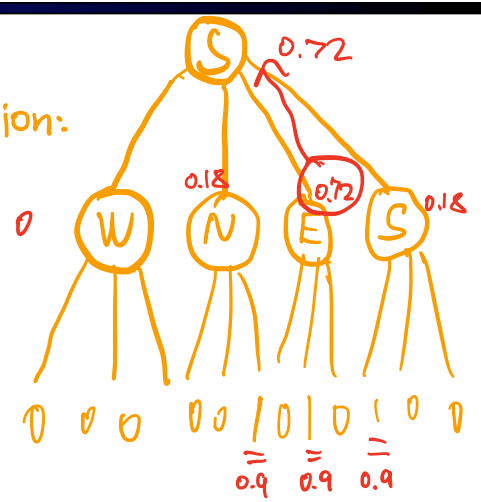

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=1



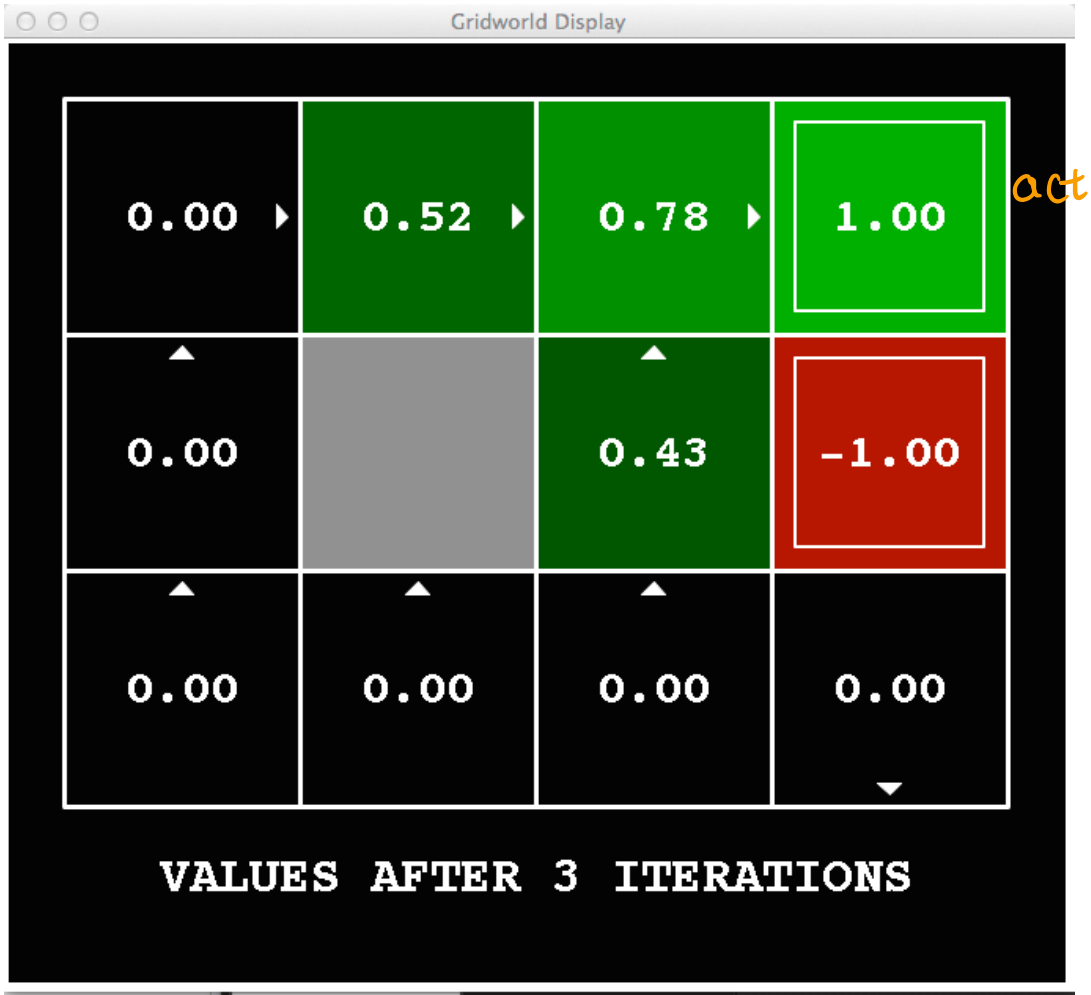

Noise = 0.2
Discount = 0.9
Living reward = 0

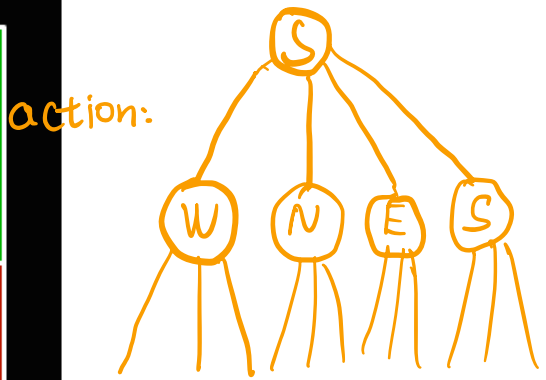
# k=2


Noise = 0.2
Discount = 0.9
Living reward = 0

# k=3




0 0 0.72 0.72 0.72 1 0 0

0.72 1 0 1 0

0.72 x 0.8 + 0.1 x 1

0.72 x 0.1 + 1 x 0.8

0.072 + 0.8

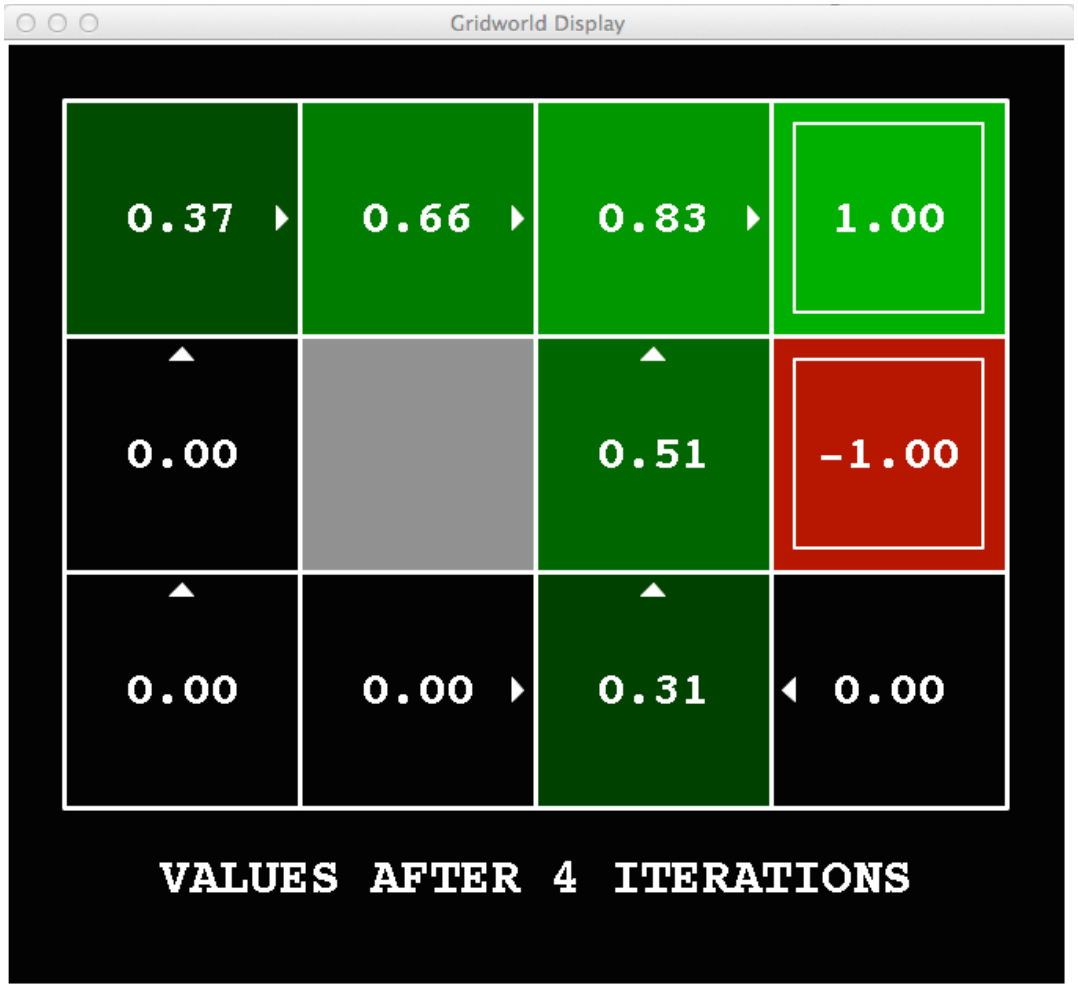
= 0.7848

Noise = 0.2

Discount = 0.9

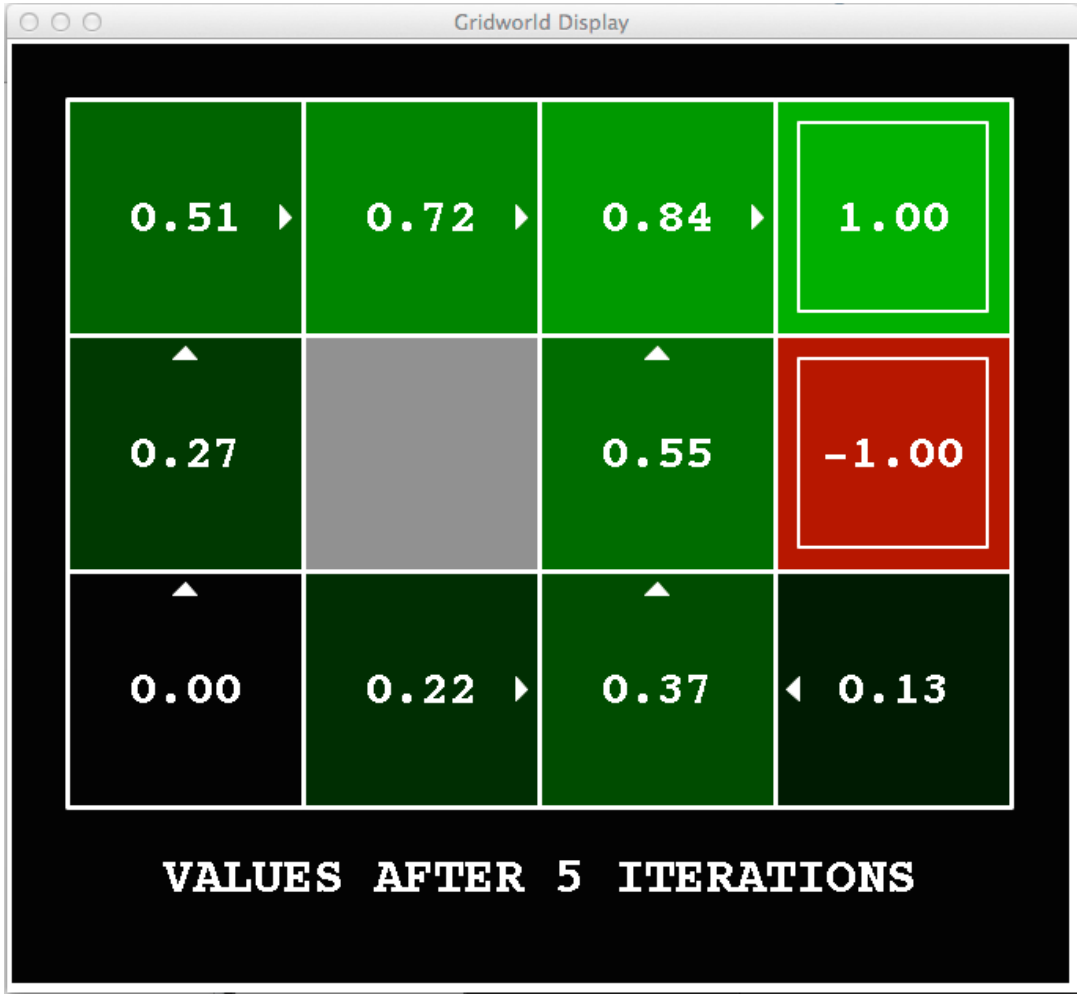
Living reward = 0

# k=4



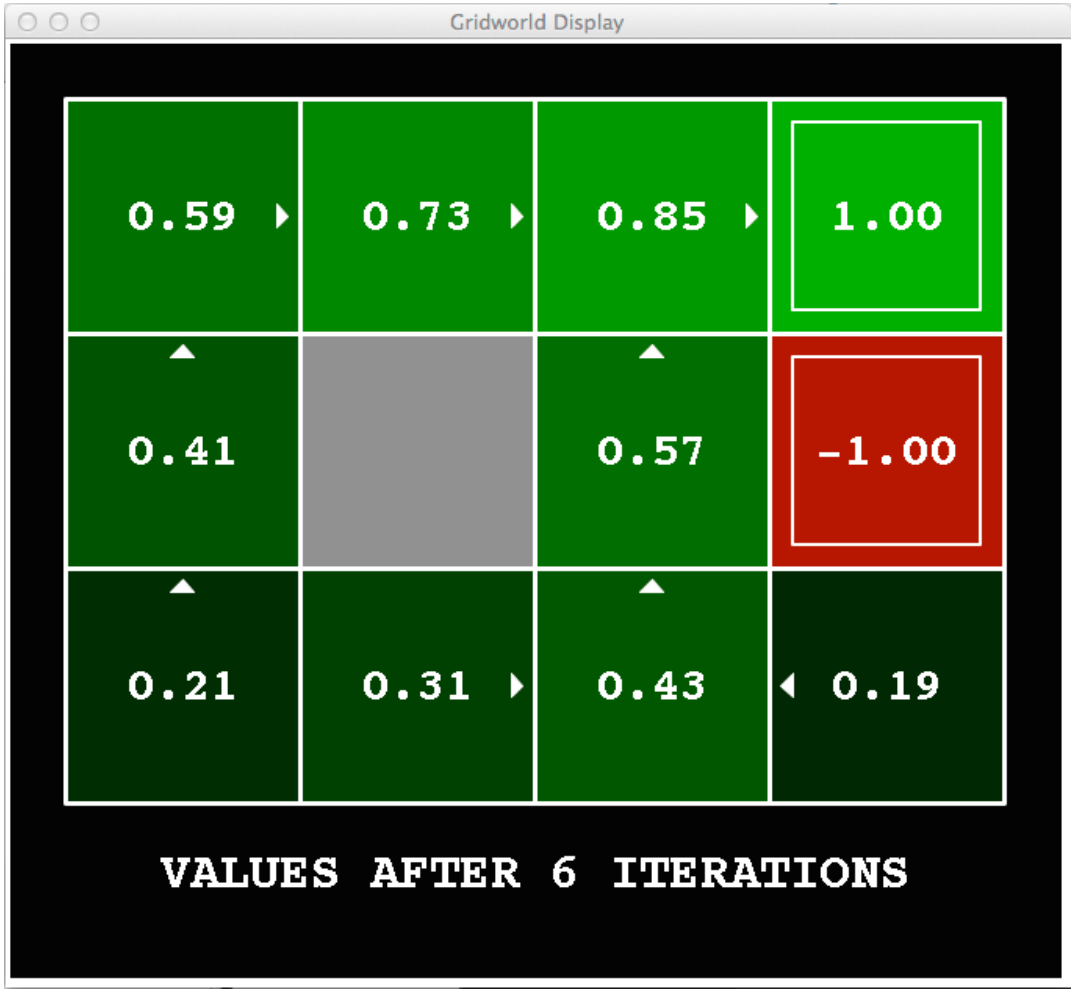

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=5



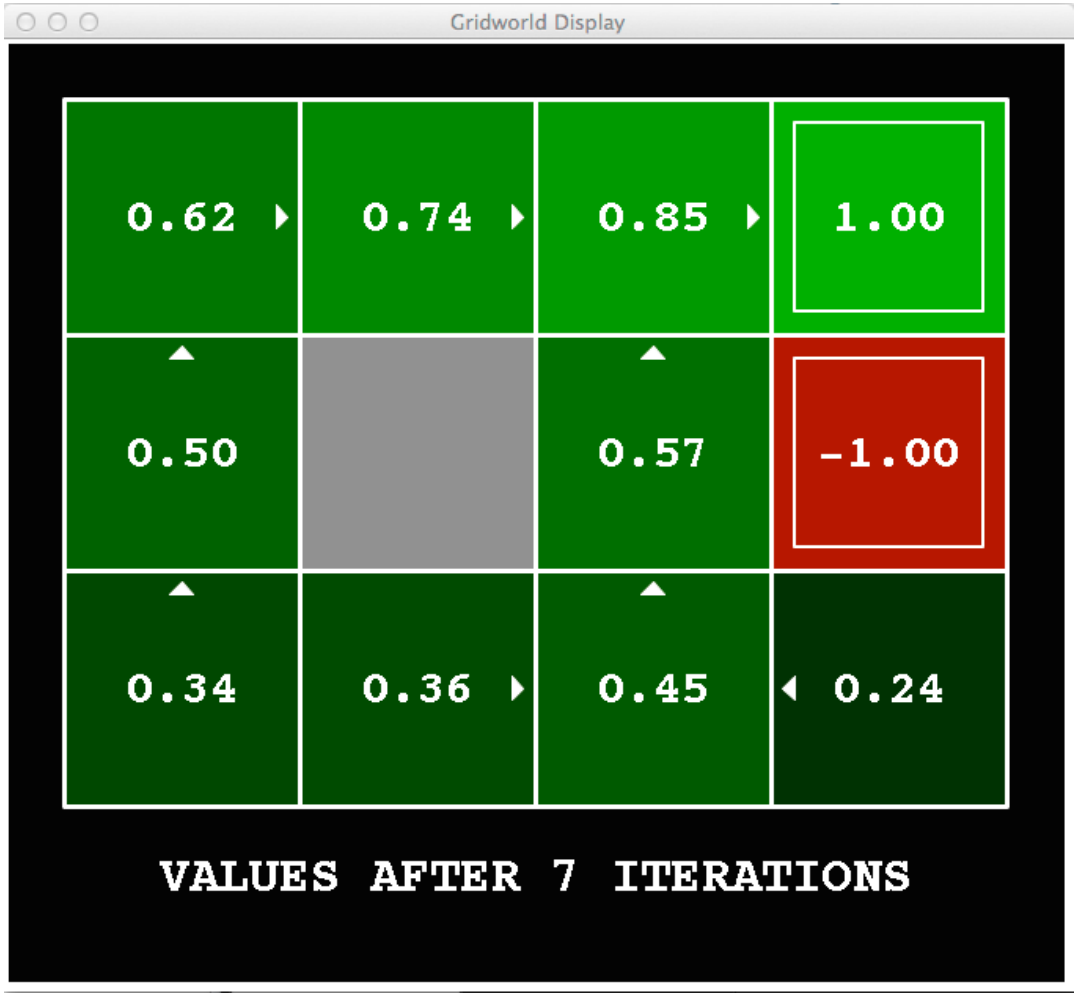

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=6



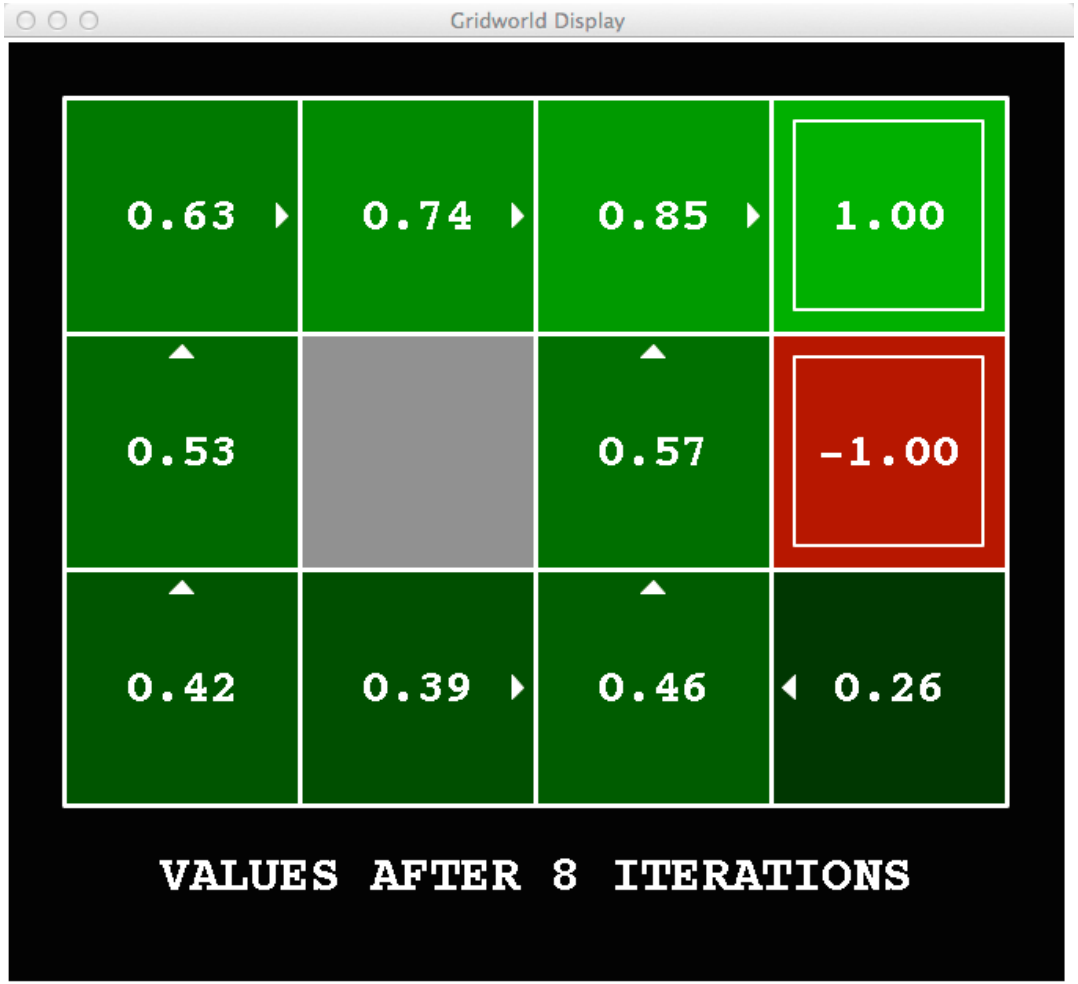

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=7



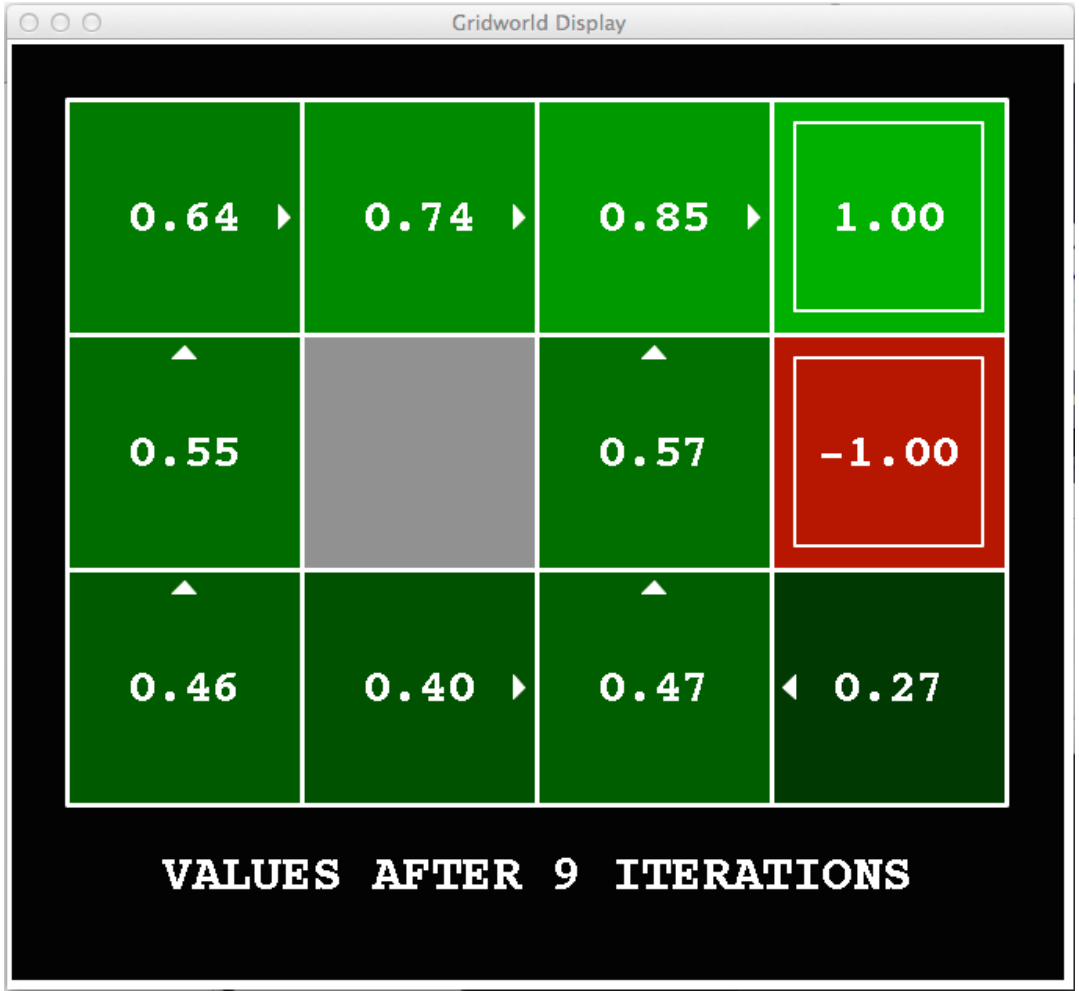

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=8



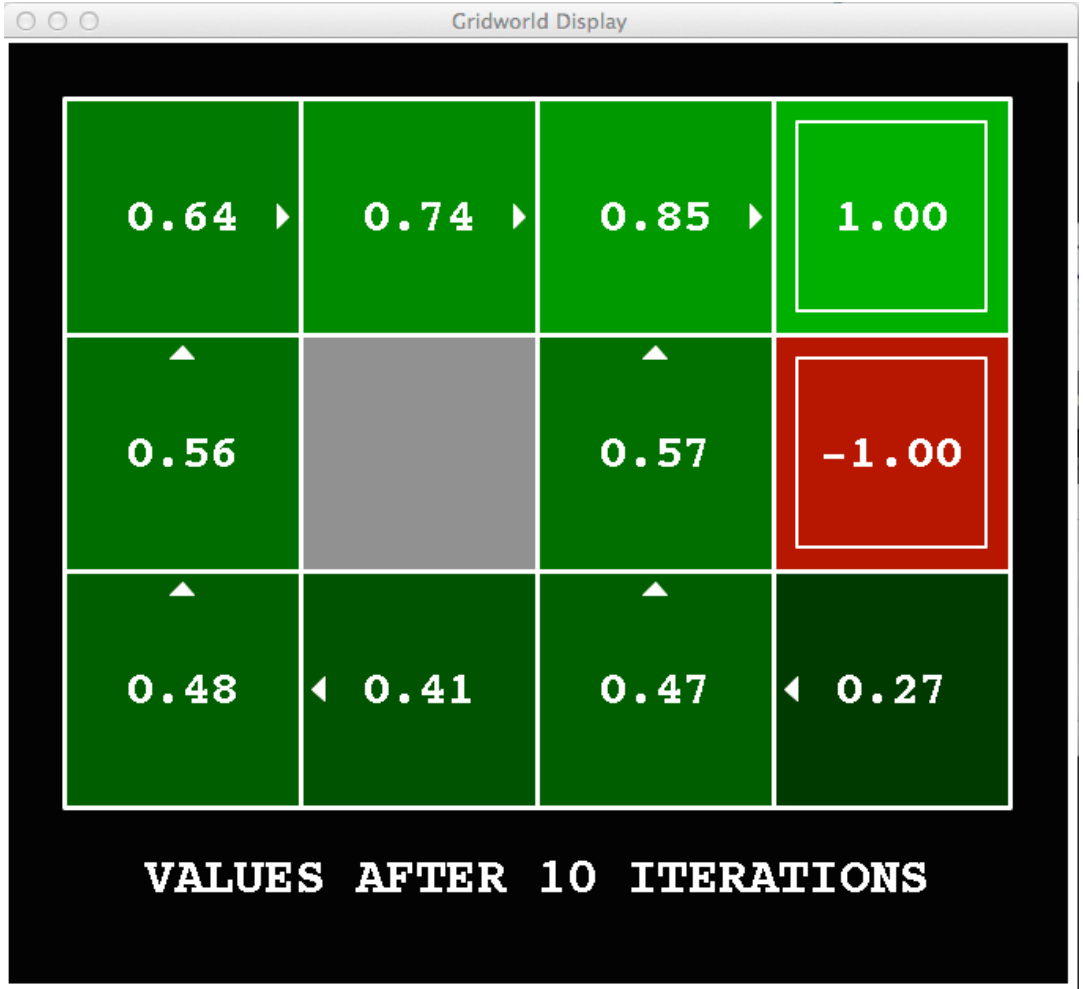

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=9



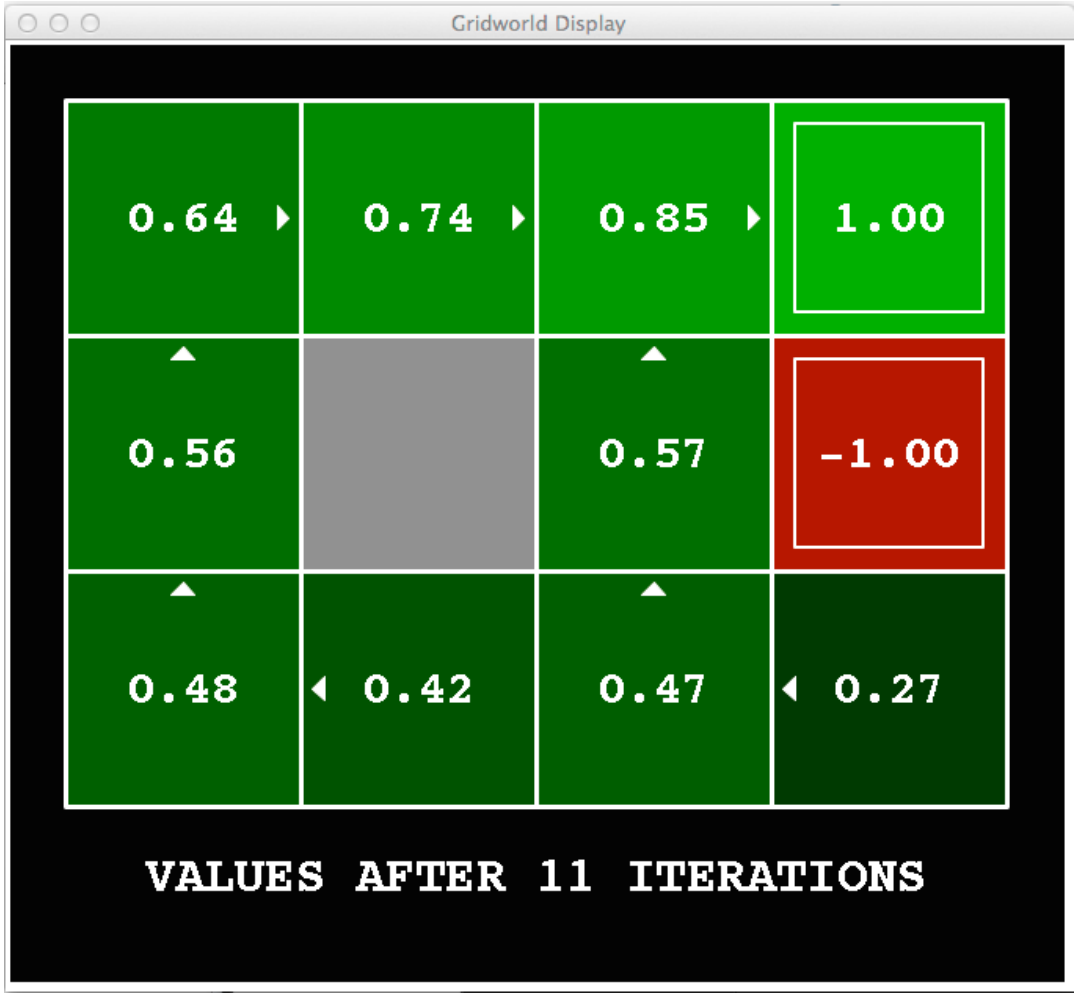

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=10



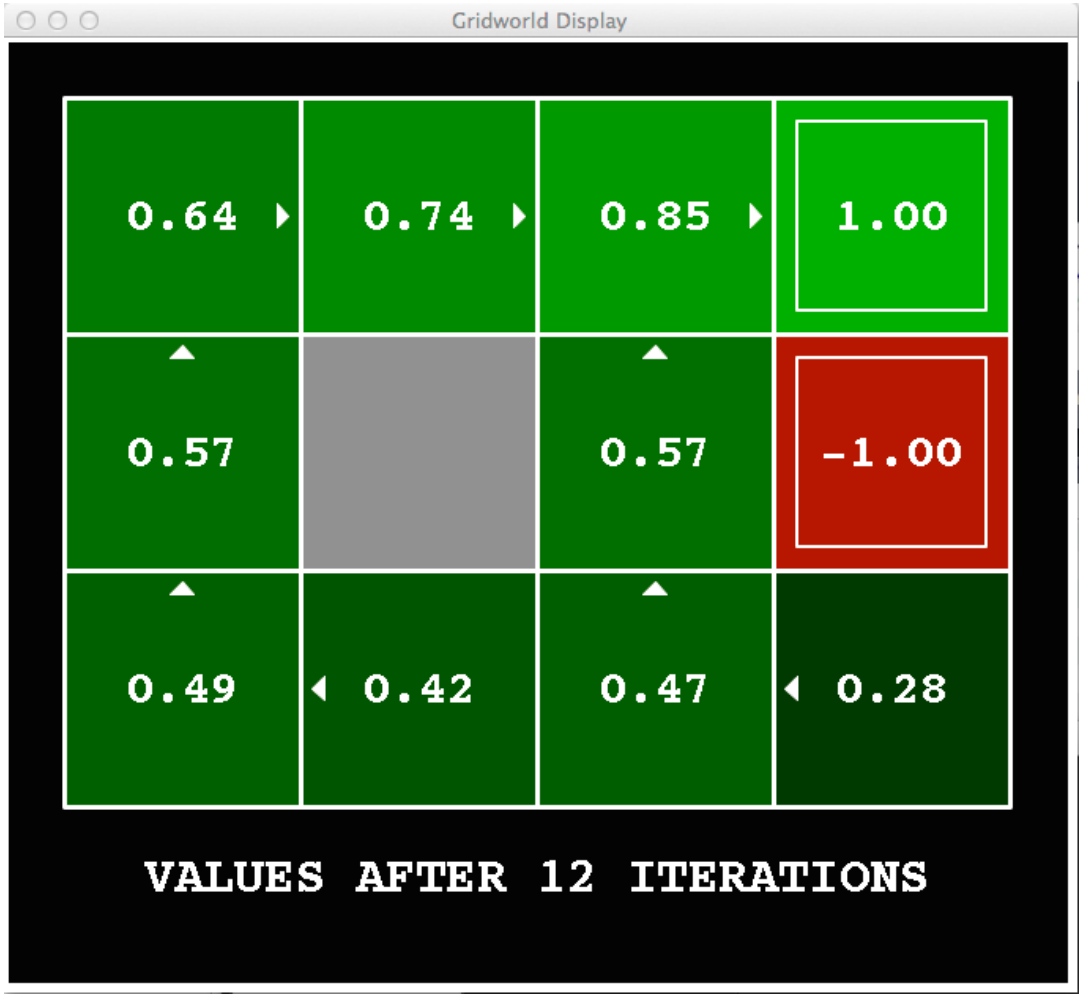

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=11




Noise = 0.2
Discount = 0.9
Living reward = 0

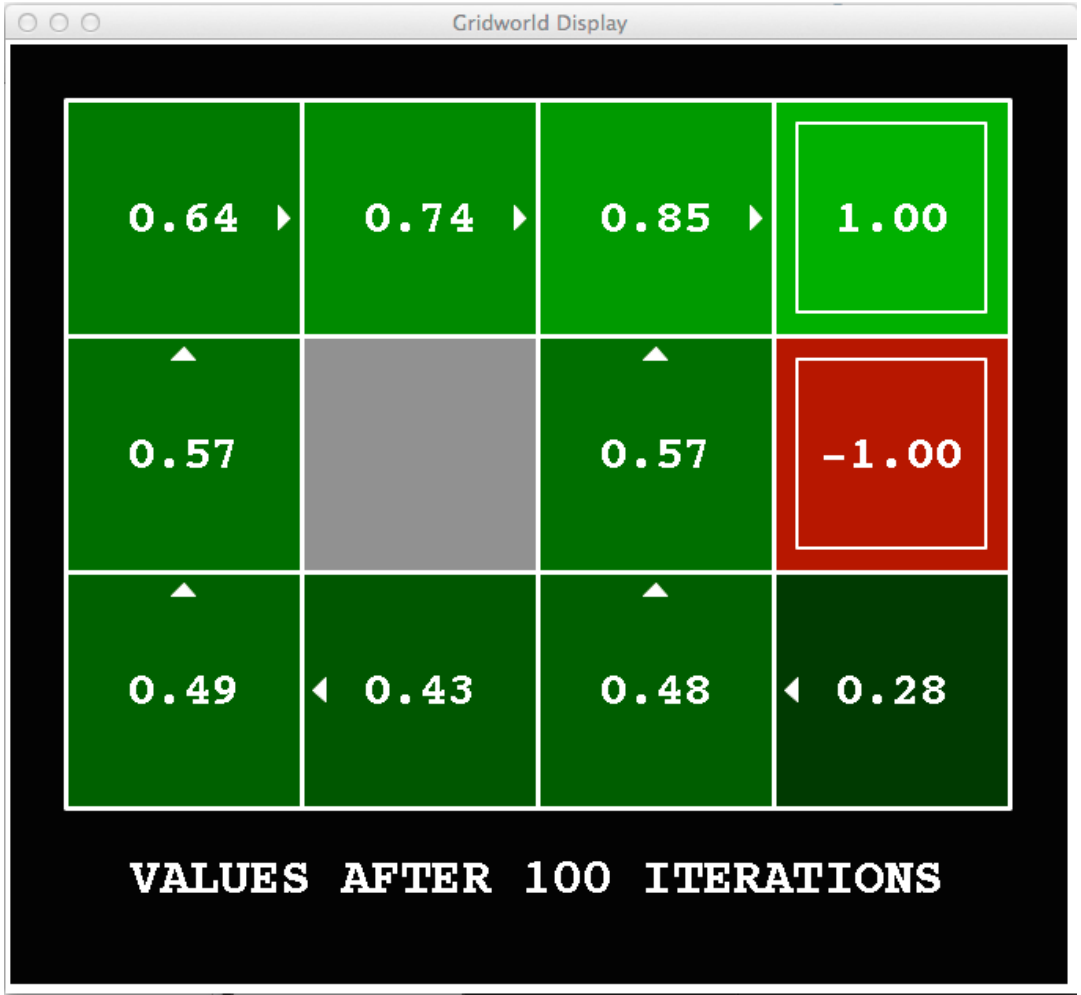
# k=12




Action 基本上没有改变了

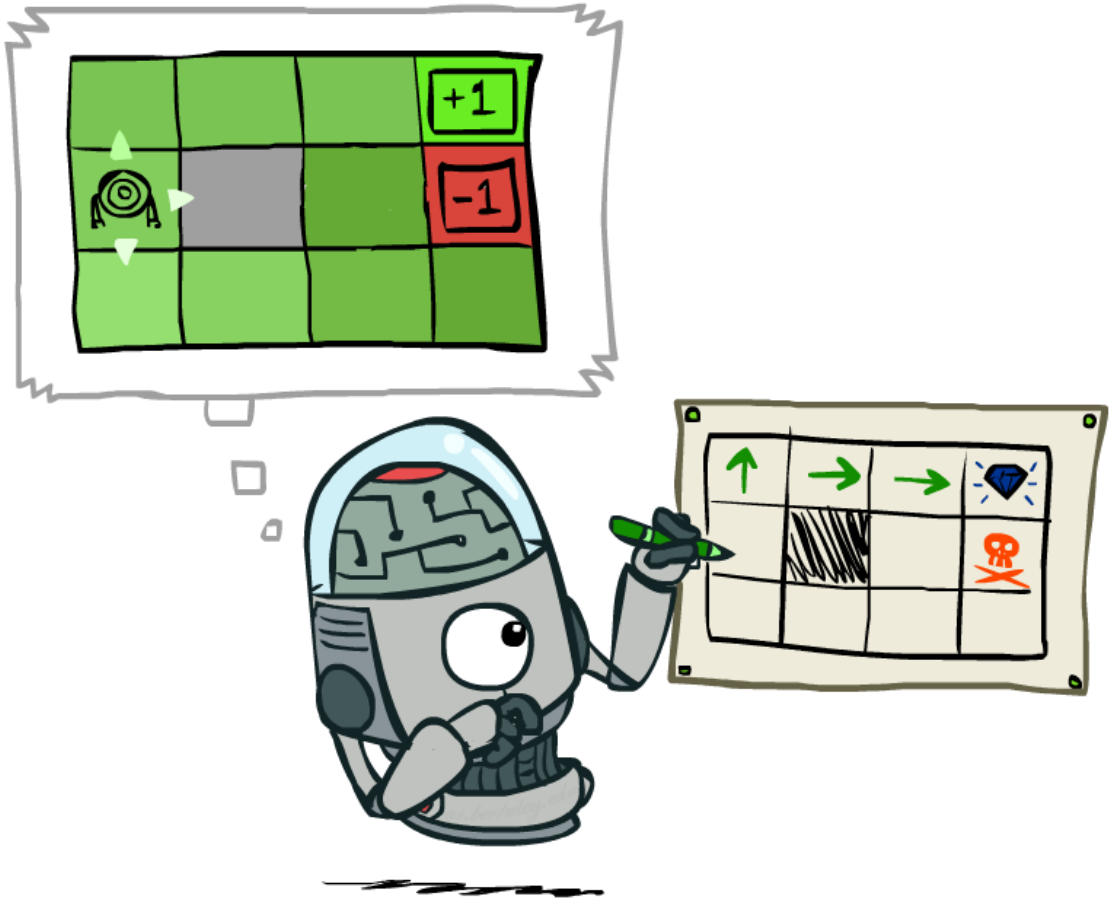
Noise = 0.2
Discount = 0.9
Living reward = 0

# k=100



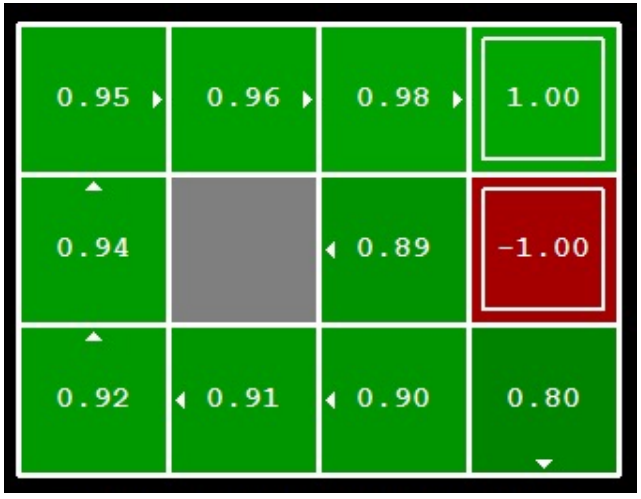

Noise = 0.2
Discount = 0.9
Living reward = 0

# Policy Extraction



# Computing Actions from Values

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



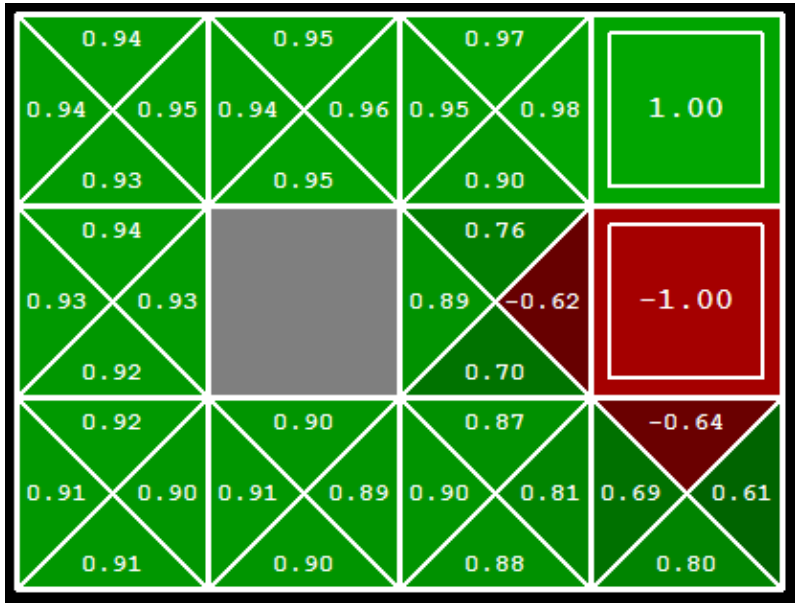

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

策略的提取

# 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: actions are easier to select from q-values than values!
- Q-values can also be computed in value iteration

# Q-Value Iteration

- Value iteration: find successive (depth-limited) values
  - Start with $V_0(s) = 0$
  - Given $V_k$, calculate the depth k+1 values for all states:

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

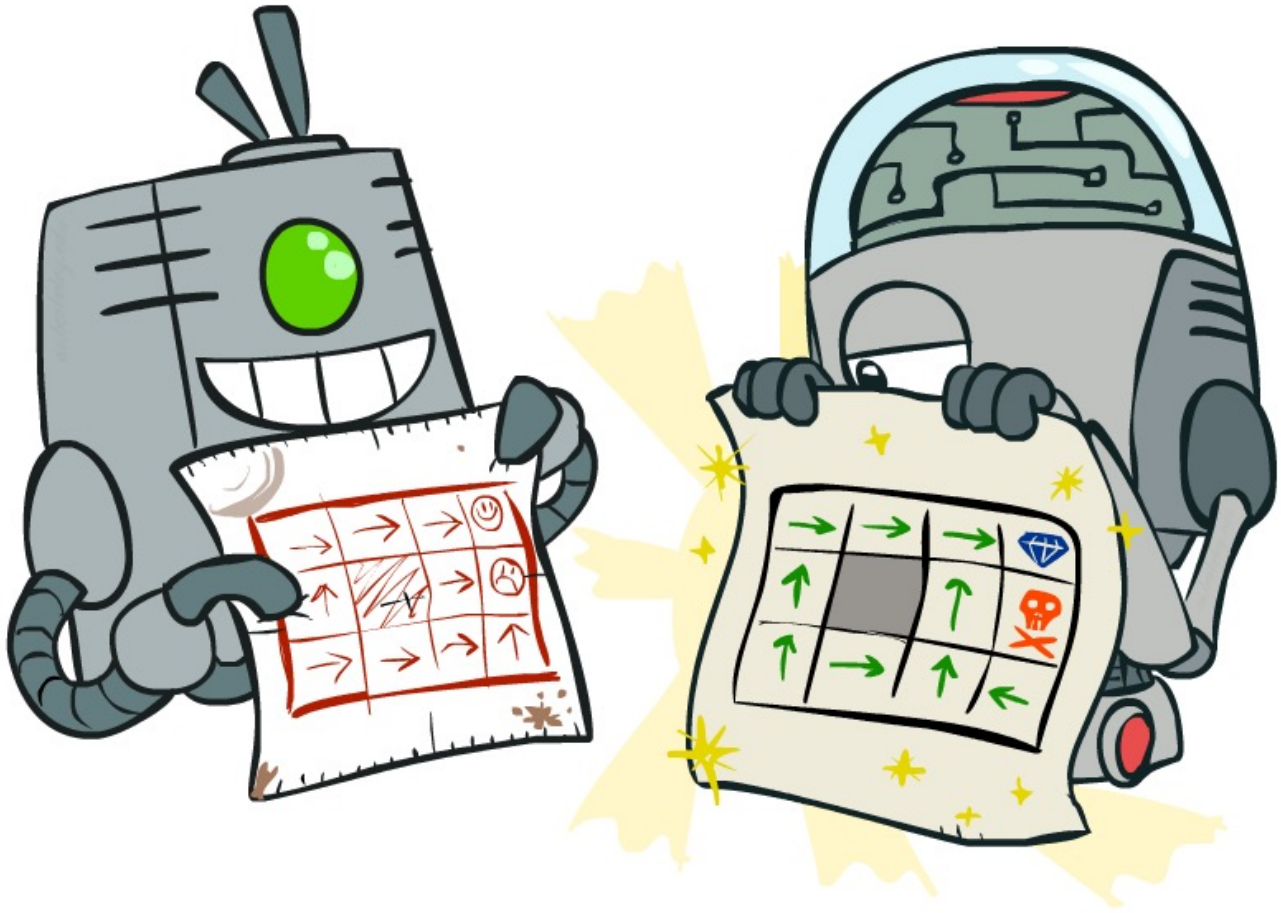
$Q_{k+1}(s,a)$

$\max_{a'} Q_k(s',a')$

- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s,a) = 0$
  - Given $Q_k$, calculate the depth k+1 q-values for all q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

# Policy Methods

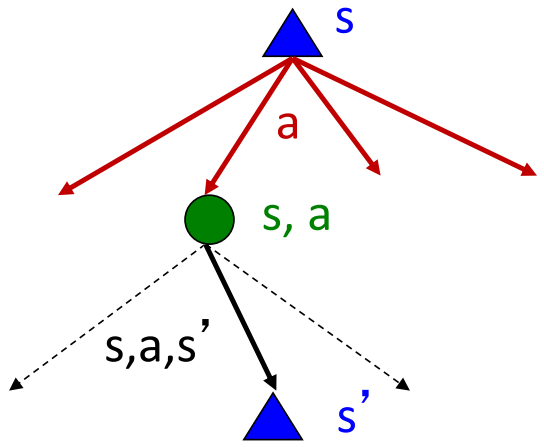


# Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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

实际上是一个期望




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

X: 解空间特别特别大

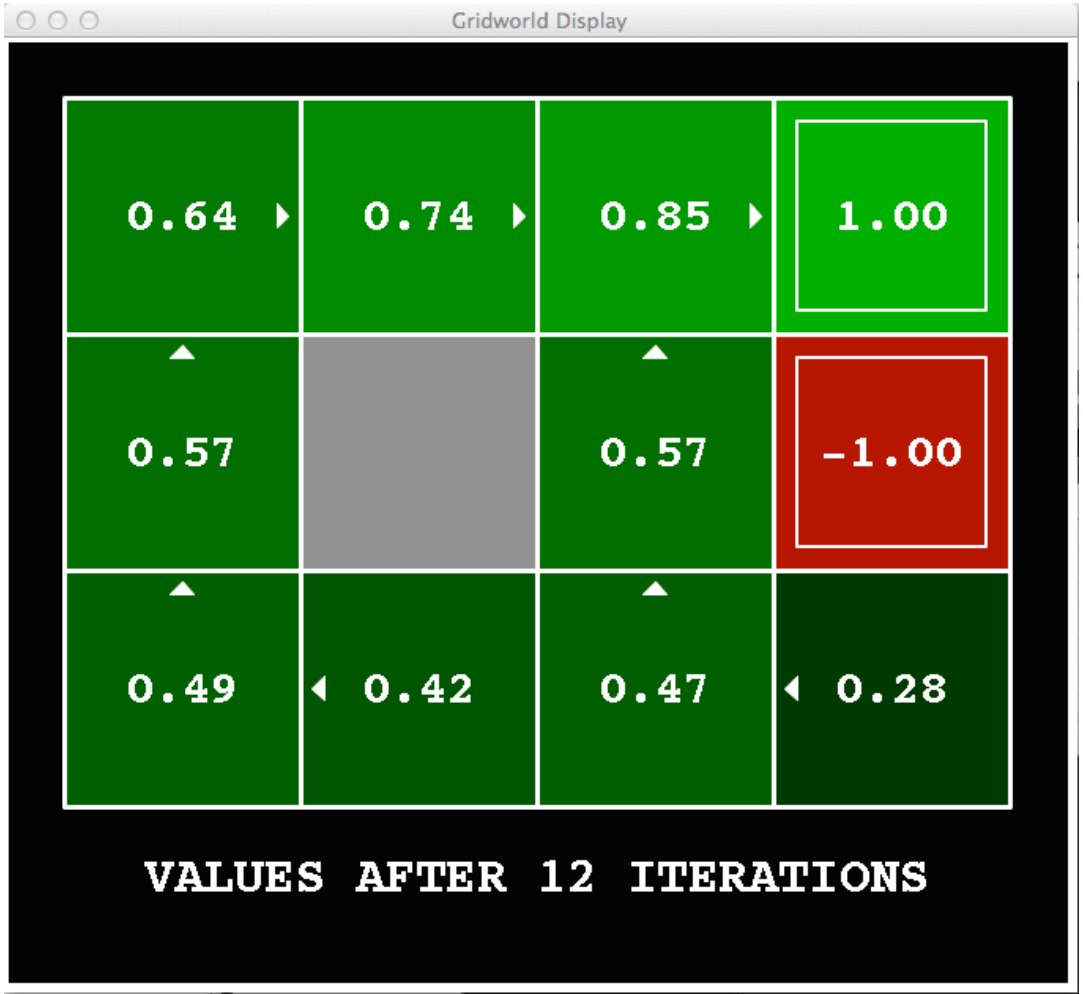
- Problem 2: The "max" at each state rarely changes
  - The policy often converges long before the values

方向(action)比value先确定.

能否直接显式得到policy

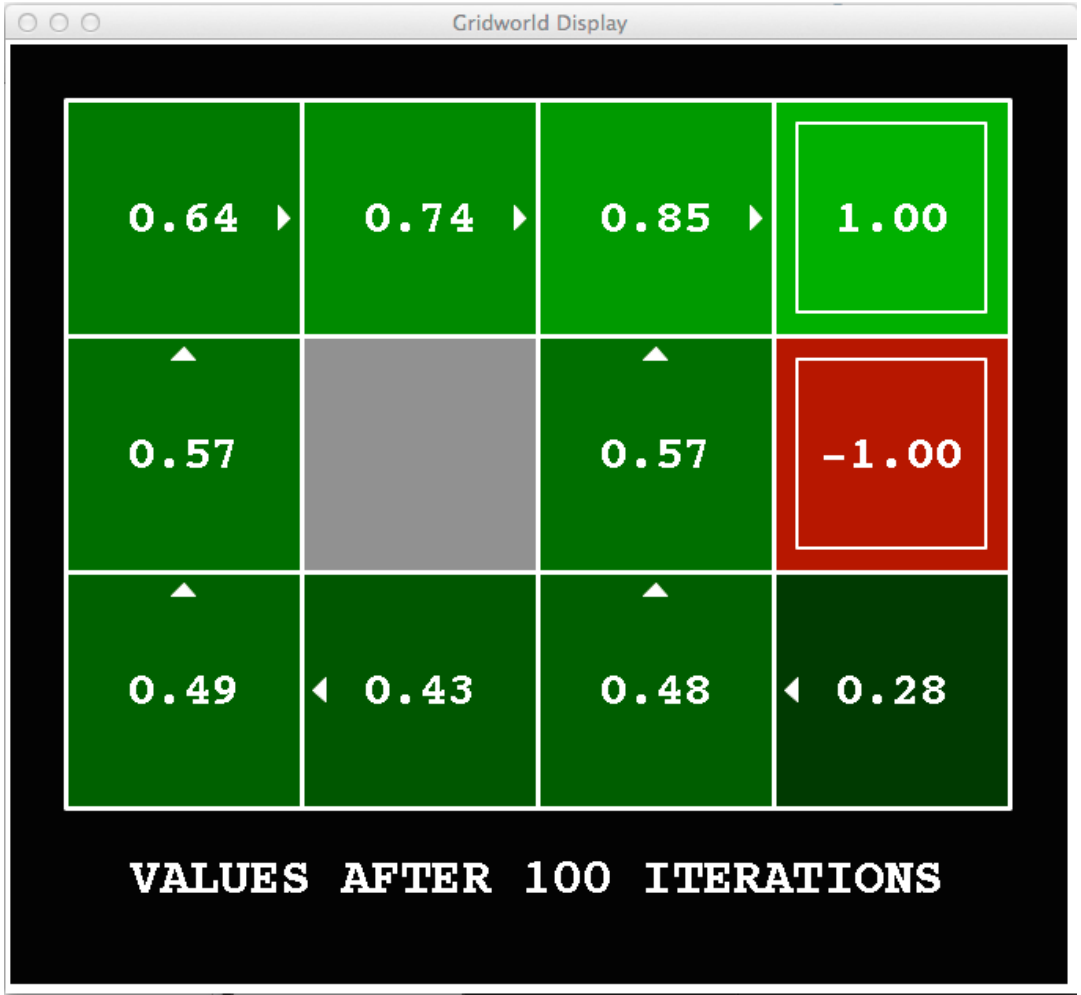
我们应该关注policy

# k=12




Noise = 0.2
Discount = 0.9
Living reward = 0

# k=100




如果大幅度:
1) 采样不够准.
2) 随机性很强.

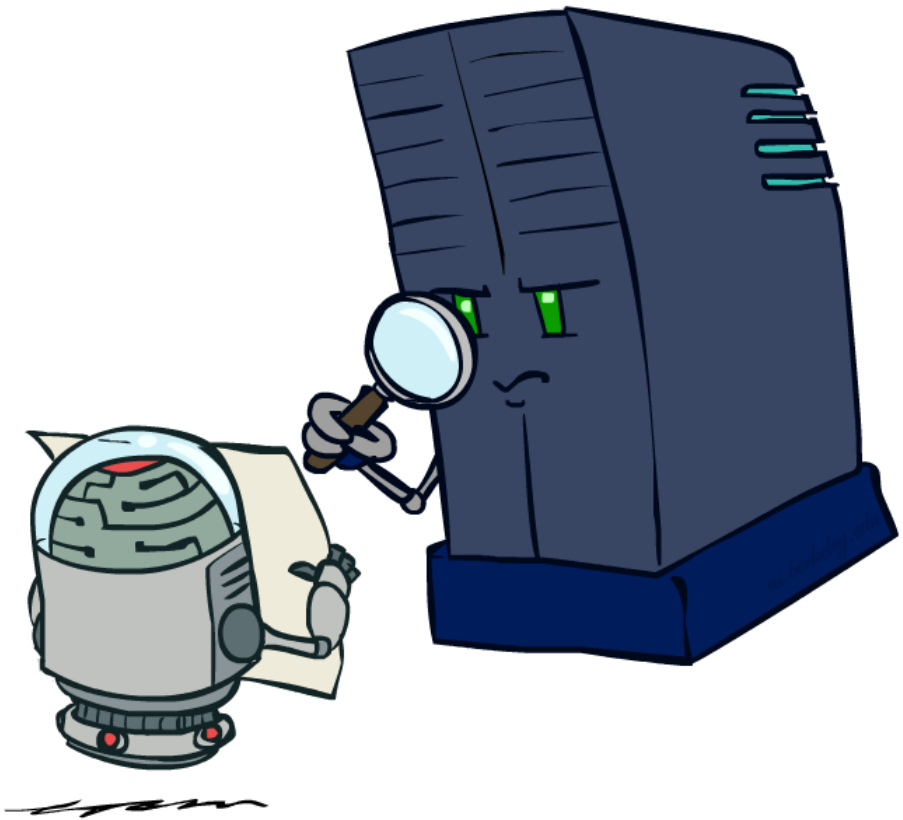
Noise = 0.2
Discount = 0.9
Living reward = 0

# Policy Iteration

- Policy iteration: an alternative approach for value iteration
  - Step 1: Policy evaluation: calculate utilities for some fixed (not optimal) policy
  - 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

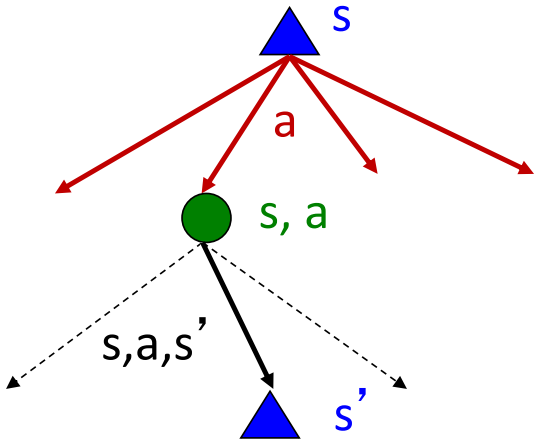
- It's still optimal!
- Can converge (much) faster under some conditions

# Step 1: Policy Evaluation



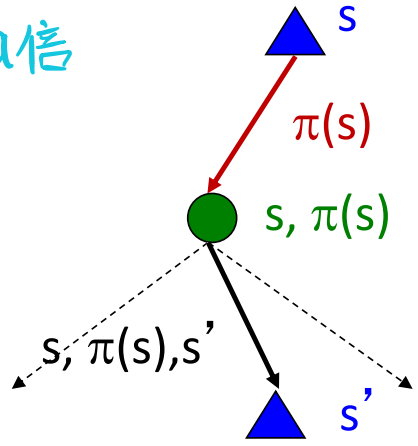
# Fixed Policies 评测的是固定的Policy

Do the optimal action




解空间缩小a倍

Do what π says to do




(只take一个action)

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

# Utilities for a Fixed Policy

- The utility of a state s, under a fixed policy $\pi$:

  $V^{\pi}(s)$ = expected utility starting in s and following $\pi$

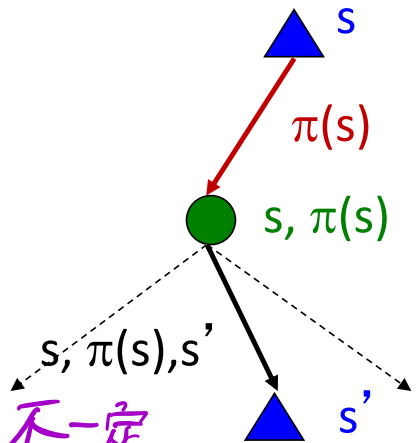
  follow: policy of $\pi$

- Recursive relation (one-step look-ahead):

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

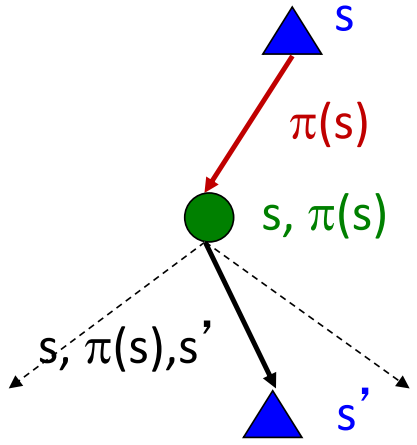
↳ 根据某个固定策略得到的值，不一定最优

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



# Policy Evaluation

- How do we calculate the values under a fixed policy $\pi$?




- Idea 1: Iterative updates (like value iteration)
  - Start with $V_0^\pi(s) = 0$
  - Given $V_k^\pi$, calculate the depth k+1 values for all states:

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

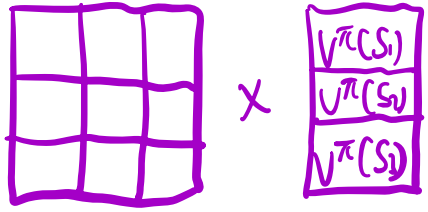
  - Repeat until convergence
  - Efficiency: $O(S^2)$ per iteration

固定了a: π(s)

- Idea 2: Without the maxes, the Bellman equations are just a linear system

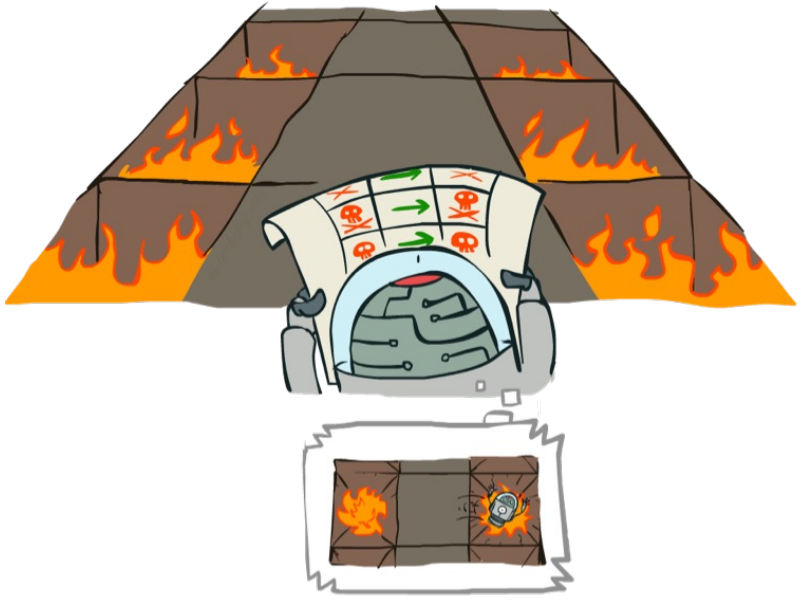
$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

  - Solvable with a linear system solver



# Example: Policy Evaluation

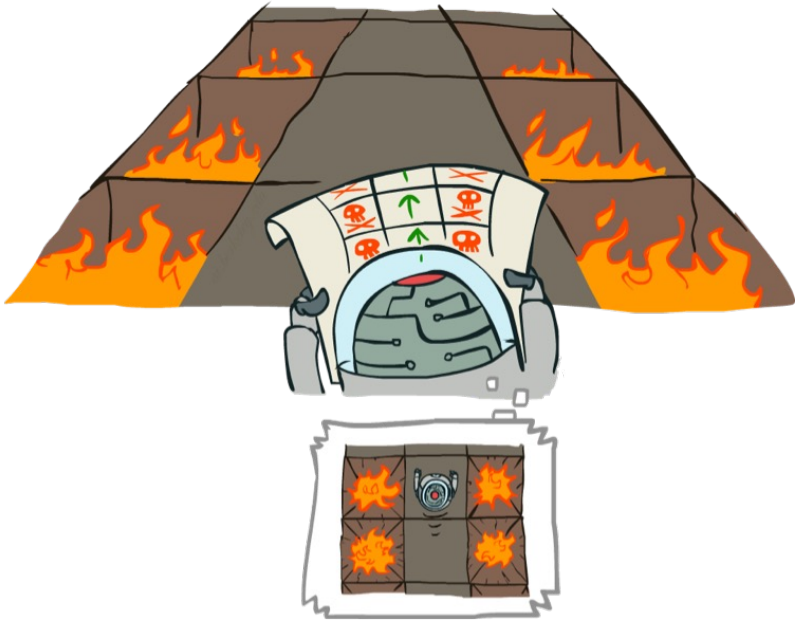
Always Go Right



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

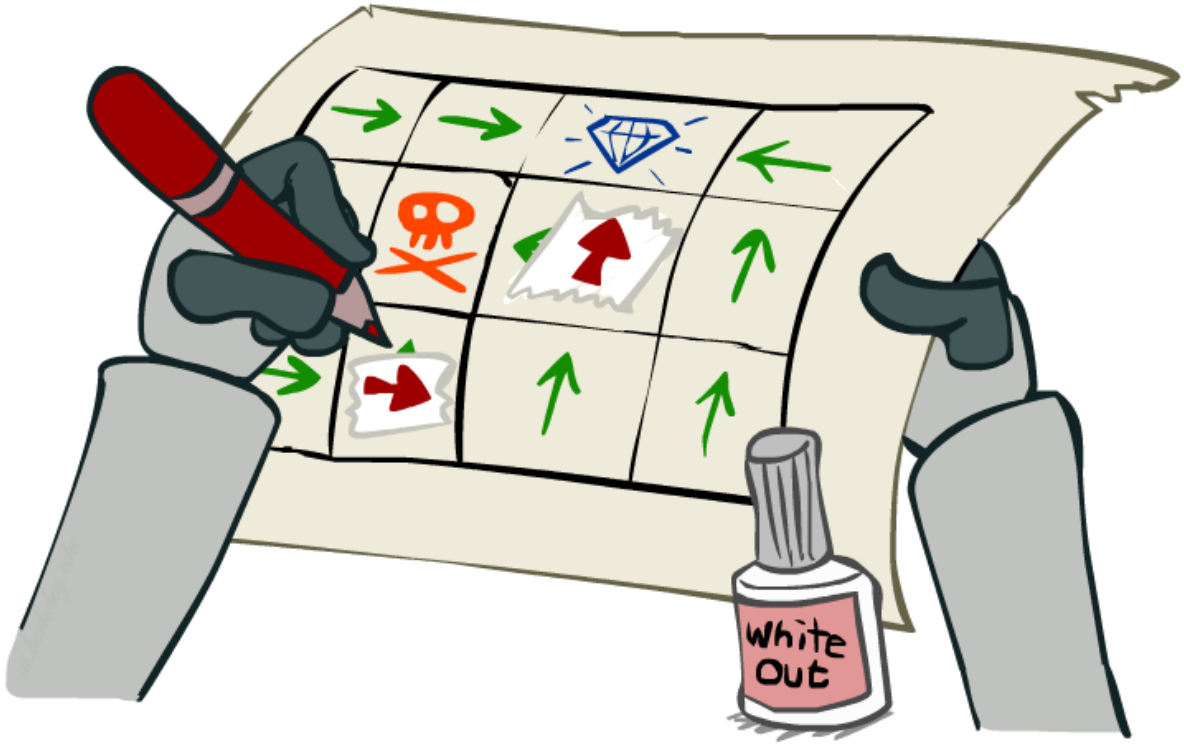
# Example: Policy Evaluation

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 |

# Step 2: Policy Improvement



# Policy Improvement

$V^{\pi_i}$ 我是在step1已知了

- Step 2: Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

至少执行了一次最优动作

$\pi_i$: optimal policy in i

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

optimal policy in i+1

$\hookrightarrow V^a_{k+1}(s)$

- Policy Iteration: repeat the two steps until policy converges

使policy更新为当前状态下最大的value

$S^2$

# Policy Iteration

- Initialize $\pi_0(s) = some\ default\ action$ for all s
- for i:
  - Initialize $V_0^{\pi_i}(s) = 0$ for all s
  - for k:

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

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

保证其在收敛

对于$\pi_i$，固定$\pi_i$，计算 $V^{\pi_i}(S)$ { $V^{\pi_i}(S_1)$, $V^{\pi_i}(S_2)$, ..., $V^{\pi_i}(S_n)$

对于$\pi_{i+1}(S)$ 则在$Q(S,a)$中取Max对应的a，

在i中最优的action

That's another way you could have, you could have written this out.
这是另一种你本可以写出来的方式。

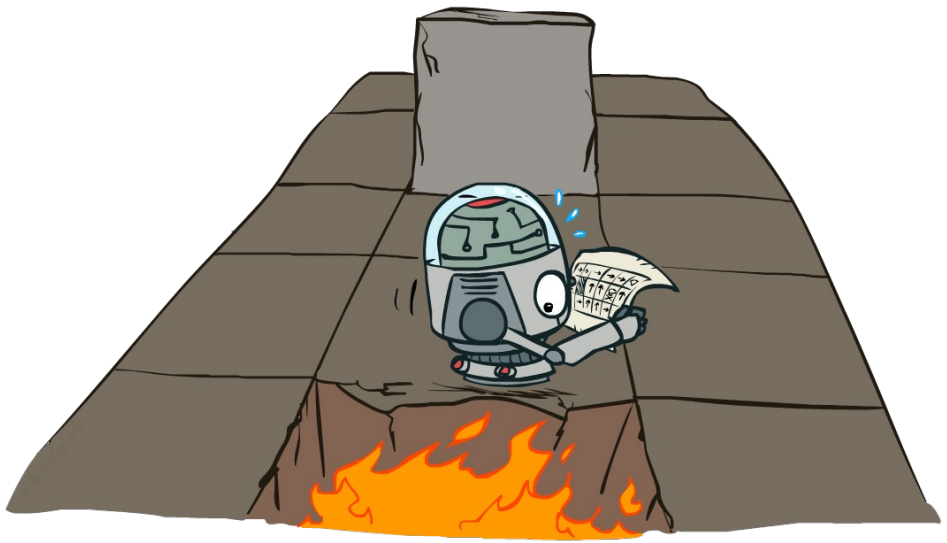
核心：Policy收敛更快

# 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)
  - May converge faster
- Both are dynamic programs for solving MDPs

# Summary

- Markov Decision Process
  - States S, Actions A, Transitions P(s'|s,a), Rewards R(s,a,s')
- Quantities:
  - Policy, Utility, Values, Q-Values
- Solve MDP
  - Value iteration
  - Policy iteration



# CS 188 Introduction to Artificial Intelligence
# Fall 2023 Note 9

Author (all other notes): Nikhil Sharma

Author (Bayes' Nets notes): Josh Hug and Jacky Liang, edited by Regina Wang

Author (Logic notes): Henry Zhu, edited by Peyrin Kao

Credit (Machine Learning and Logic notes): Some sections adapted from the textbook *Artificial Intelligence: A Modern Approach*.

Last updated: August 26, 2023

## Policy Iteration

Value iteration can be quite slow. At each iteration, we must update the values of all $|S|$ states (where $|n|$ refers to the cardinality operator), each of which requires iteration over all $|A|$ actions as we compute the Q-value for each action. The computation of each of these Q-values, in turn, requires iteration over each of the $|S|$ states again, leading to a poor runtime of $O(|S|^2|A|)$. Additionally, when all we want to determine is the optimal policy for the MDP, value iteration tends to do a lot of overcomputation since the policy as computed by policy extraction generally converges significantly faster than the values themselves. The fix for these flaws is to use **policy iteration** as an alternative, an algorithm that maintains the optimality of value iteration while providing significant performance gains. Policy iteration operates as follows:

1. Define an *initial policy*. This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until convergence:

   就是为了计算出固定 action 的收敛值

   - Evaluate the current policy with **policy evaluation**. For a policy $\pi$, policy evaluation means computing $U^{\pi}(s)$ for all states $s$, where $U^{\pi}(s)$ is expected utility of starting in state $s$ when following $\pi$:

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

     Define the policy at iteration $i$ of policy iteration as $\pi_i$. Since we are fixing a single action for each state, we no longer need the max operator which effectively leaves us with a system of $|S|$ equations generated by the above rule. Each $U^{\pi_i}(s)$ can then be computed by simply solving this system. Alternatively, we can also compute $U^{\pi_i}(s)$ by using the following update rule until convergence, just like in value iteration:

     $$U^{\pi_i}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma U^{\pi_i}_{k}(s')]$$

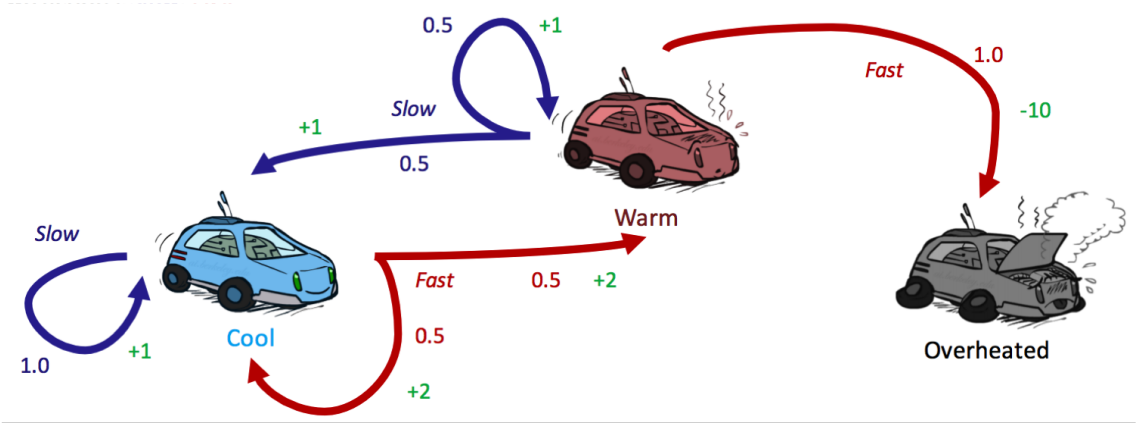
     However, this second method is typically slower in practice.
   - Once we've evaluated the current policy, use **policy improvement** to generate a better policy. Policy improvement uses policy extraction on the values of states generated by policy evaluation

to generate this new and improved policy:

$$\pi_{i+1}(s) = \underset{a}{\text{argmax}} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U^{\pi_i}(s')]$$

If $\pi_{i+1} = \pi_i$, the algorithm has converged, and we can conclude that $\pi_{i+1} = \pi_i = \pi^*$.

Let's run through our racecar example one last time (getting tired of it yet?) to see if we get the same policy using policy iteration as we did with value iteration. Recall that we were using a discount factor of $\gamma = 0.5$.




We start with an initial policy of *Always go slow*:

| | cool | warm | overheated |
|---|---|---|---|
| $\pi_0$ | *slow* | *slow* | – |

Because terminal states have no outgoing actions, no policy can assign a value to one. Hence, it's reasonable to disregard the state *overheated* from consideration as we have done, and simply assign $\forall i,\ U^{\pi_i}(s) = 0$ for any terminal state $s$. The next step is to run a round of policy evaluation on $\pi_0$:

$$\begin{aligned} U^{\pi_0}(cool) &= 1 \cdot [1 + 0.5 \cdot U^{\pi_0}(cool)] \\ U^{\pi_0}(warm) &= 0.5 \cdot [1 + 0.5 \cdot U^{\pi_0}(cool)] + 0.5 \cdot [1 + 0.5 \cdot U^{\pi_0}(warm)] \end{aligned}$$

Solving this system of equations for $U^{\pi_0}(cool)$ and $U^{\pi_0}(warm)$ yields:

| | cool | warm | overheated |
|---|---|---|---|
| $U^{\pi_0}$ | 2 | 2 | 0 |

We can now run policy extraction with these values:

$$\begin{aligned} \pi_1(cool) &= \text{argmax}\{slow : 1 \cdot [1 + 0.5 \cdot 2],\ fast : 0.5 \cdot [2 + 0.5 \cdot 2] + 0.5 \cdot [2 + 0.5 \cdot 2]\} \\ &= \text{argmax}\{slow : 2,\ fast : 3\} \\ &= \boxed{fast} \\ \pi_1(warm) &= \text{argmax}\{slow : 0.5 \cdot [1 + 0.5 \cdot 2] + 0.5 \cdot [1 + 0.5 \cdot 2],\ fast : 1 \cdot [-10 + 0.5 \cdot 0]\} \\ &= \text{argmax}\{slow : 3,\ fast : -10\} \\ &= \boxed{slow} \end{aligned}$$

Running policy iteration for a second round yields $\pi_2(cool) = fast$ and $\pi_2(warm) = slow$. Since this is the same policy as $\pi_1$, we can conclude that $\pi_1 = \pi_2 = \pi^*$. Verify this for practice!

| | **cool** | **warm** |
|---|---|---|
| $\pi_0$ | *slow* | *slow* |
| $\pi_1$ | *fast* | *slow* |
| $\pi_2$ | *fast* | *slow* |

This example shows the true power of policy iteration: with only two iterations, we've already arrived at the optimal policy for our racecar MDP! This is more than we can say for when we ran value iteration on the same MDP, which was still several iterations from convergence after the two updates we performed.

# Summary

The material presented above has much opportunity for confusion. We covered value iteration, policy iteration, policy extraction, and policy evaluation, all of which look similar, using the Bellman equation with subtle variation. Below is a summary of the purpose of each algorithm:

- *Value iteration*: Used for computing the optimal values of states, by iterative updates until convergence.
- *Policy evaluation*: Used for computing the values of states under a specific policy.
- *Policy extraction*: Used for determining a policy given some state value function. If the state values are optimal, this policy will be optimal. This method is used after running value iteration, to compute an optimal policy from the optimal state values; or as a subroutine in policy iteration, to compute the best policy for the currently estimated state values.
- *Policy iteration*: A technique that encapsulates both policy evaluation and policy extraction and is used for iterative convergence to an optimal policy. It tends to outperform value iteration, by virtue of the fact that policies usually converge much faster than the values of states.