CS 573: Artificial Intelligence

Markov Decision Processes

Dan Weld

University of Washington

Many slides by Dan Klein & Pieter Abbeel / UC Berkeley. (http://ai.berkeley.edu) and by Mausam & Andrey Kolobov

Logistics

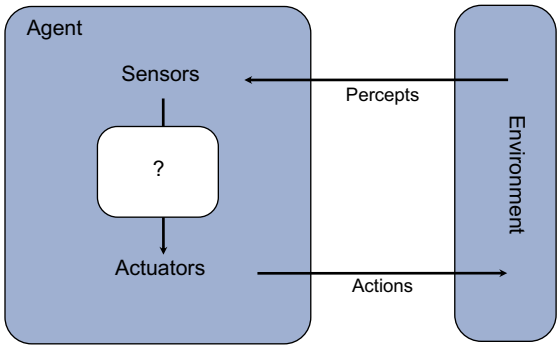
- PS2 due on Sat
- Midterm 1 next Thurs
  - Cover material through and including today
  - Closed book, no internet, no calculator
  - You may bring one 8.5 x 11" piece of paper with notes
  - Email me & TAs by 2/4 if you will be OOT and need a makeup

Outline

- State Spaces
- Search Algorithms & Heuristics
- Adversarial Games
- Stochastic "Games"
  - Expectimax
  - Markov Decision Processes
    - Value iteration
    - Policy iteration
  - Reinforcement Learning

Agent *vs.* Environment

- An agent is an entity that perceives and acts.
- A rational agent selects actions that ***maximize its utility function.***



Deterministic *vs.* ***stochastic***
Fully observable *vs.* partially observable

Axioms of Rational Preferences

Orderability

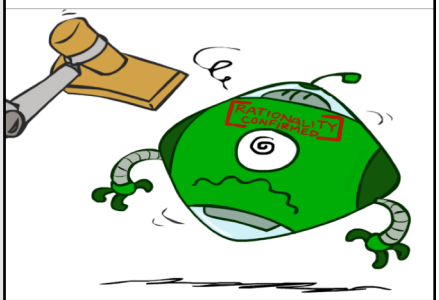
$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p\ [p, A;\ 1-p, C] \sim B$$



Axioms of Rational Preferences

Orderability

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p\ [p, A;\ 1-p, C] \sim B$$

Substitutability

$$A \sim B \Rightarrow [p, A;\ 1-p, C] \sim [p, B; 1-p, C]$$

Axioms of Rational Preferences

Orderability

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p \; [p, A; \; 1-p, C] \sim B$$

Substitutability

$$A \sim B \Rightarrow [p, A; \; 1-p, C] \sim [p, B; 1-p, C]$$

Monotonicity

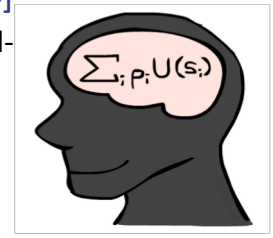
$$A \succ B \Rightarrow$$
$$(p \geq q \Leftrightarrow [p, A; \; 1-p, B] \succeq [q, A; \; 1-q, B])$$

MEU Principle

- Theorem [Ramsey, 1931; von Neumann & Morgenstern, 1944]
  - Given any preferences satisfying these constraints, there exists a real-valued function U such that:

$$U(A) \geq U(B) \;\; \Leftrightarrow \;\; A \succeq B$$

$$U([p_1, S_1; \; \ldots \; ; \; p_n, S_n]) = \sum_i p_i U(S_i)$$



  - I.e. values assigned by U preserve preferences of both prizes and lotteries!
- Maximum expected utility (MEU) principle:
  - Choose the action that maximizes expected utility
  - Note: an agent can be entirely rational (consistent with MEU) without ever representing or manipulating utilities and probabilities
  - E.g., a lookup table for perfect tic-tac-toe, a reflex vacuum cleaner

Utility Scales

- How Measure ***Human*** Utility? (e.g., what units)?
  - Micromorts: one-millionth chance of death, useful for paying to reduce product risks, etc.
  - QALYs: quality-adjusted life years, useful for medical decisions involving substantial risk

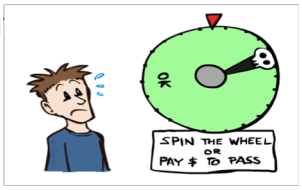
- Maximize expected utility →
  - Behavior is invariant under positive ***linear*** transformation

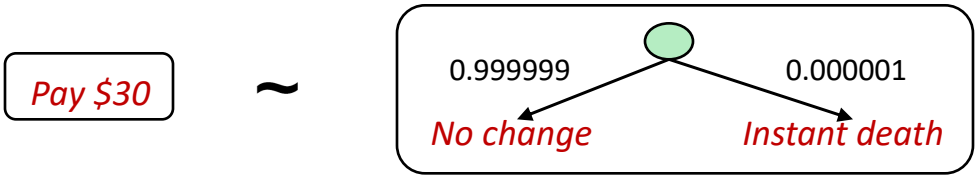
$$U'(x) = k_1 U(x) + k_2 \quad \text{where } k_1 > 0$$

- WoLoG Normalized utilities: $u_+ = 1.0$, $u_- = 0.0$

Human Utilities

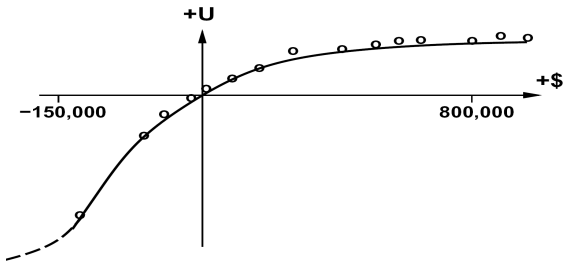
- Utilities map states to real numbers. Which numbers?
- Standard approach to assessment (elicitation) of human u
  - Compare a prize A to a standard lottery L_p between
    - "best possible prize" u_+ with probability p
    - "worst possible catastrophe" u_- with probability 1-p
  - Adjust lottery probability p until indifference: A ~ L_p
  - Resulting p is a utility in [0,1]





Money

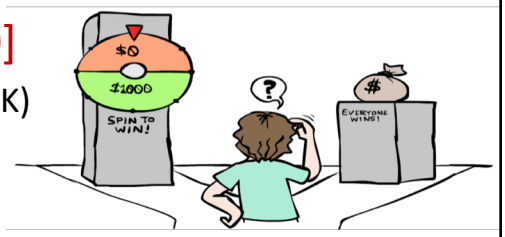
- Money ***does not*** behave as a utility function, but we can talk about the utility of having money (or being in debt)
- Given a lottery L = [p, $X; (1-p), $Y]
  - The expected monetary value EMV(L) is p*X + (1-p)*Y
  - U(L) = p*U($X) + (1-p)*U($Y)
  - Typically, U(L) < U(EMV(L))
  - In this sense, people are risk-averse
  - When deep in debt, people are risk-prone



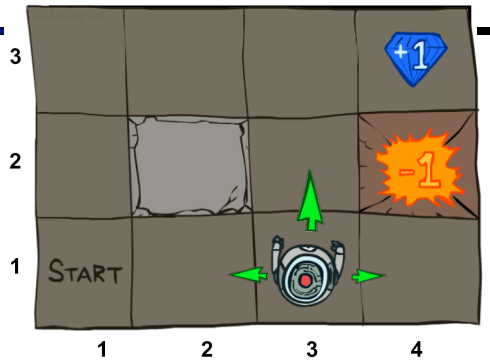
Example: Insurance

Consider the lottery [0.5, $1M; 0.5, $0]

- What is its expected monetary value? ($500K)
- What is its certainty equivalent?
  - Monetary value acceptable in lieu of lottery
  - $400K for most people
- Difference of $100K is the insurance premium
  - There's an insurance industry because people will pay to reduce their risk
  - If everyone were risk-neutral, no insurance needed!
- It's win-win: you'd rather have the $400K and
- The insurance company would rather have the lottery (why?)



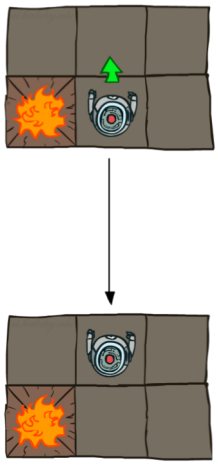
Example: Grid World



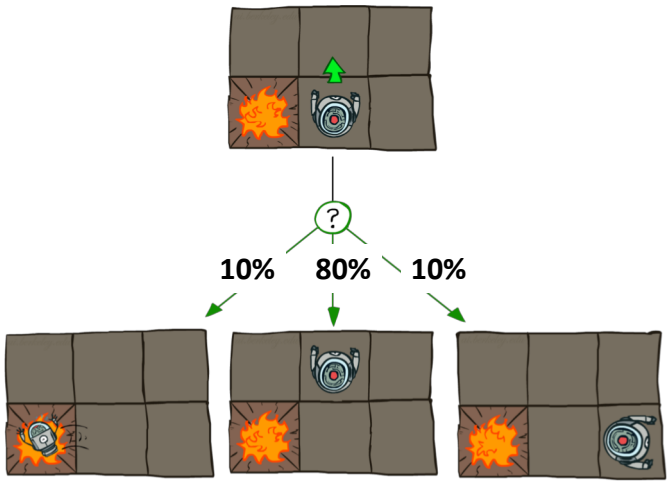
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement
  - Actions may not have intended effects
- The agent receives `rewards' on each time step
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: ~ maximize sum of rewards

Grid World Actions

Deterministic Grid World



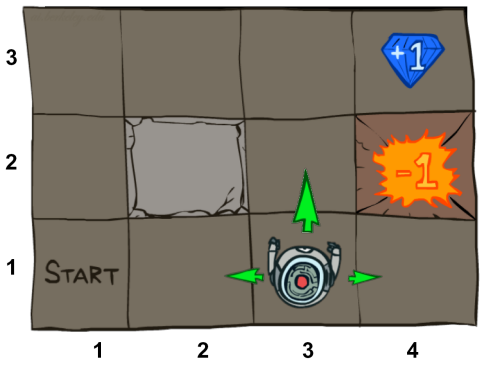
Stochastic Grid World



Assuming no wall in direction agent would have been taken

Markov Decision Processes

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



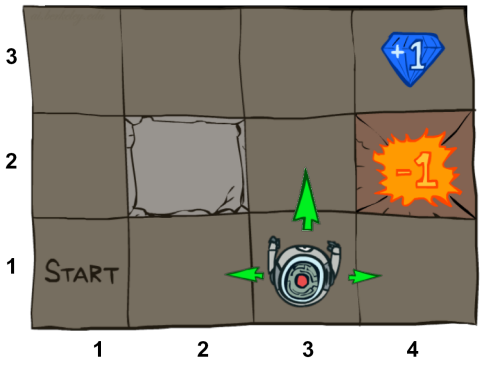
$T(s_{11}, E, \ldots)$
$\ldots$
$T(s_{31}, N, s_{11}) = 0$
$\ldots$
$T(s_{31}, N, s_{32}) = 0.8$
$T(s_{31}, N, s_{21}) = 0.1$
$T(s_{31}, N, s_{41}) = 0.1$
$\ldots$

T is a Big Table!
11 X 4 x 11 = 484 entries

For now, we give this as input to the agent

Markov Decision Processes

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



$\ldots$
$R(s_{32}, N, s_{33}) = -0.01$ ← ***Cost of breathing***
$\ldots$
$R(s_{32}, N, s_{42}) = -1.01$
$\ldots$
$R(s_{33}, E, s_{43}) = 0.99$

R is also a Big Table!

For now, we also give this to the agent

Markov Decision Processes

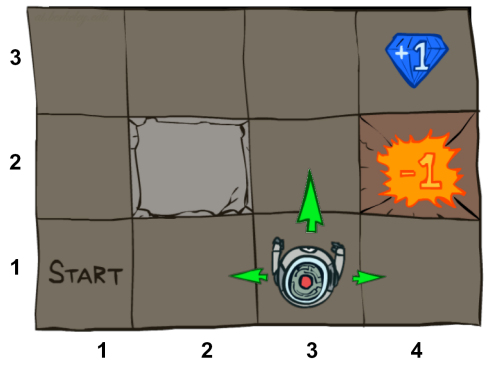
- An MDP is defined by:
  - A set of states s ∈ S
  - A set of actions a ∈ A
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s'| s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')

...

$R(s_{33}) = -0.01$

$R(s_{42}) = -1.01$

$R(s_{43}) = 0.99$



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***

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

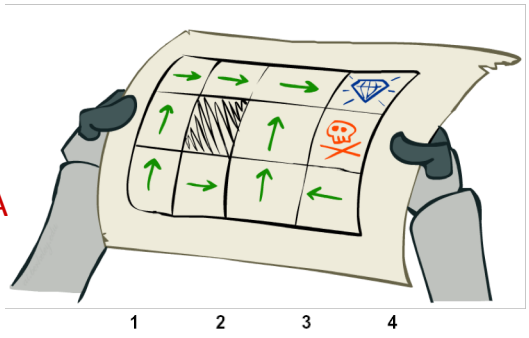
$$= P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

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

Andrey Markov
(1856-1922)

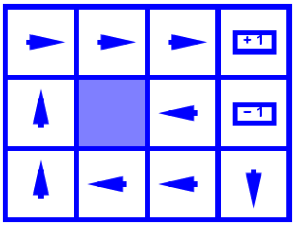
Input: MDP, Output: Policy

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal
- For MDPs, want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent
- Expectimax didn't output an entire policy
  - It computed the action for a single state only

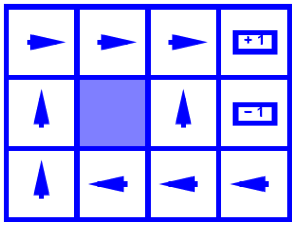


Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

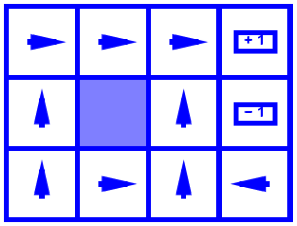
Optimal Policies



R(s) = -0.01

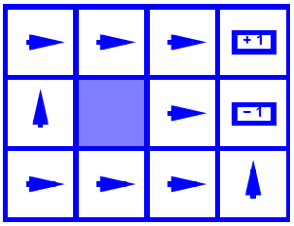


R(s) = -0.03



Cost of breathing

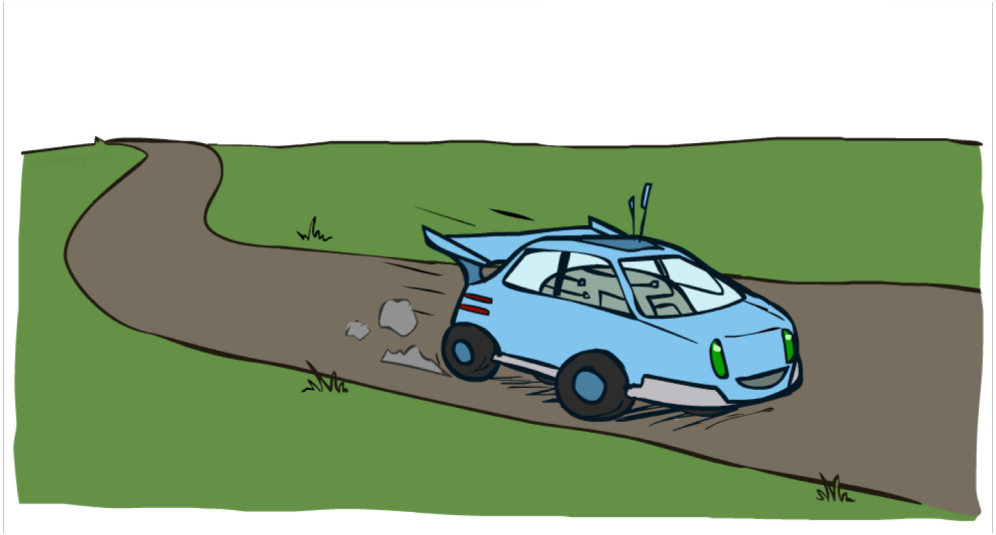
R(s) = -0.4



R(s) = -2.0

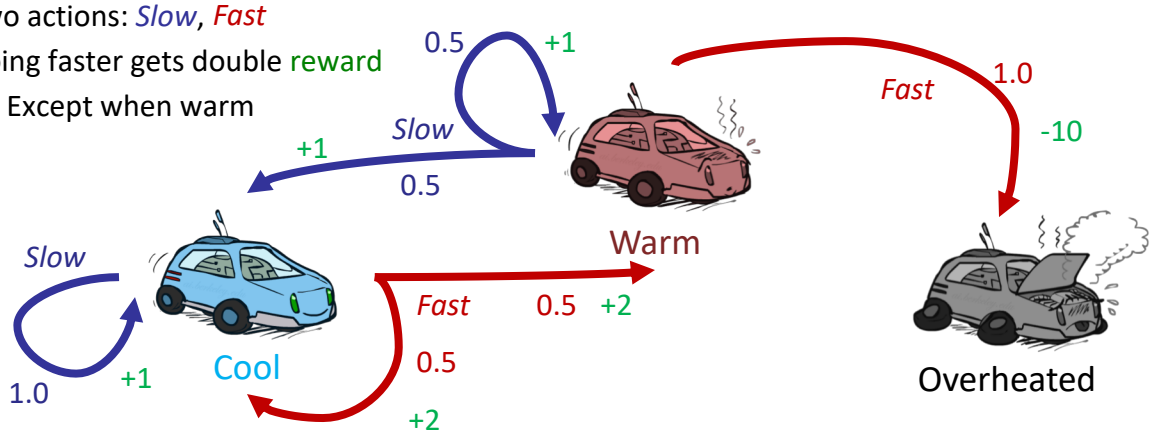
Another Example: ***Autonomous Driving***

(slightly simplified)



Example: Autonomous Driving

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



Example: Autonomous Driving

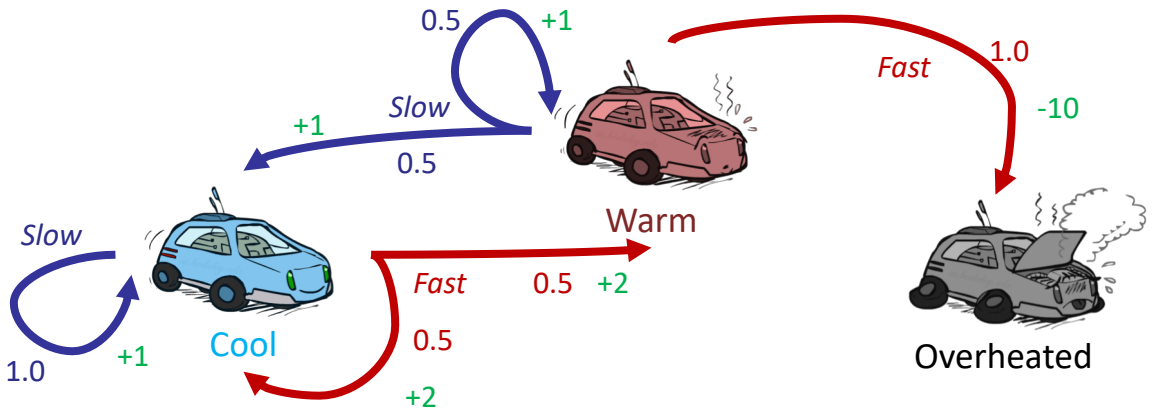
S ?

A ?

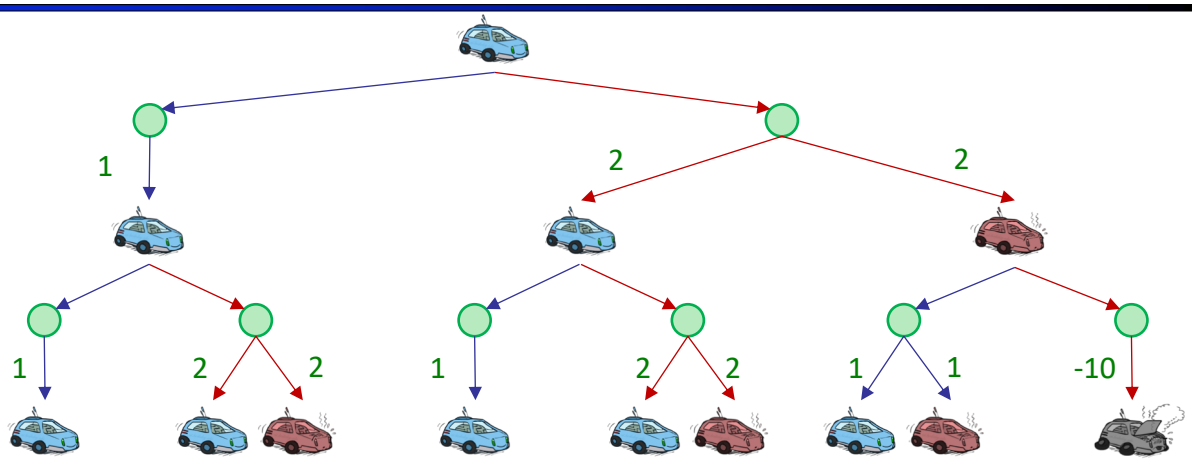
T ?

R ?

S_0 ?

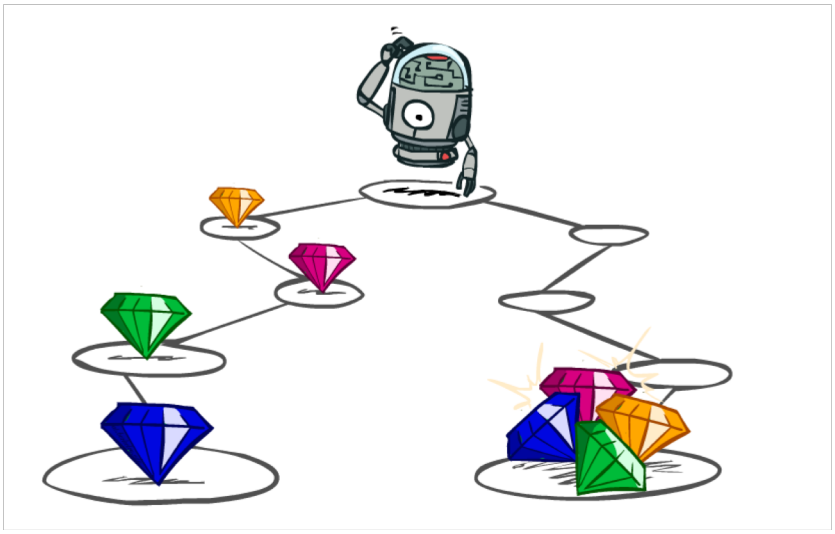


Driving: Search Tree



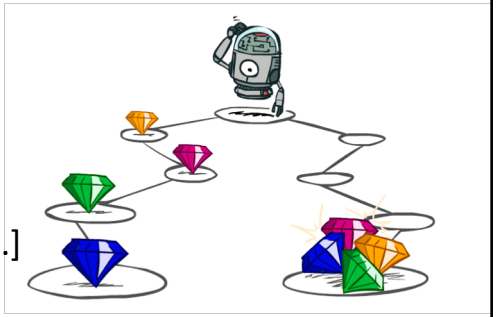
Two challenges for ExpectiMax: 1) Repeated states, 2) incremental reward
(Great solutions coming soon)

Utilities of Sequences



Utilities of Sequences

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



Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

1

Worth Now

γ

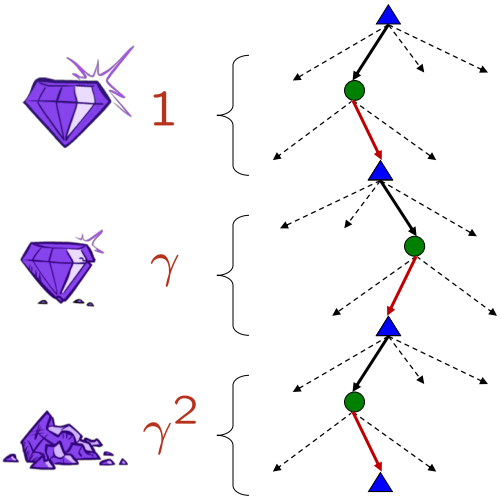
Worth Next Step

γ^2

Worth In Two Steps

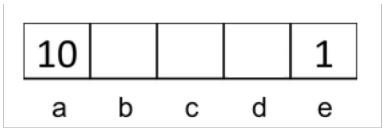
Discounting

- How to discount?
  - Each time we descend a level, we multiply by the discount
- 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 = 2.75
  - U([3,1,1]) = 1*3 + 0.5*1 + 0.25*1 = 3.75
  - U([1,2,3]) < U([3,1,1])



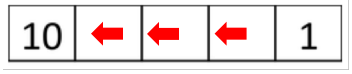
Quiz: Discounting

- Given:

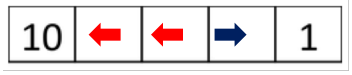


  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

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



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

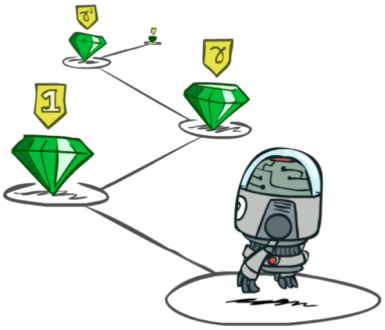


-

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

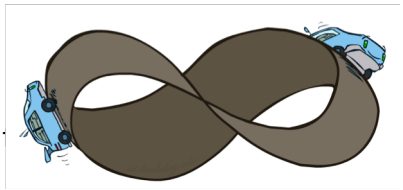
Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:

1. **Discounting:** use $0 < \gamma < 1$

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

   Smaller γ means smaller "horizon" – shorter term



2. **Finite horizon:** (similar to depth-limited search)

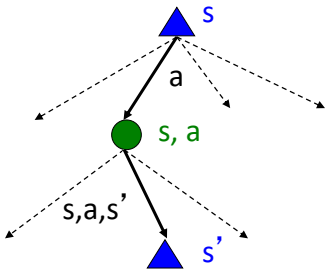
   Add utilities, but terminate episodes after a fixed T-steps lifetime

   Gives non-stationary policies (π depends on time left)!

3. **Absorbing state:** guarantee that for every policy, a terminal state (like "overheated" for racing) will eventually be reached (eg. If ***every*** action has a non-zero chance of overheating)

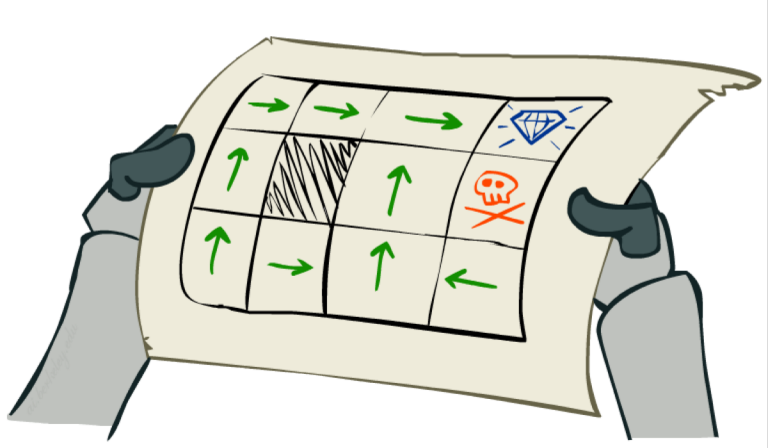
Recap: Defining MDPs

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



- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

Solving MDPs



- Value Iteration
  - Asynchronous VI
  - RTDP
  - Etc...
- Policy Iteration
- Reinforcement Learning

π^* Specifies The Optimal Policy

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

V* = Optimal Value Function

The expected value (utility) of state s:

$V^*(s)$

"expected utility starting in s & acting optimally forever"

Equivalently: "expected value of s, following π^* forever"

Q*

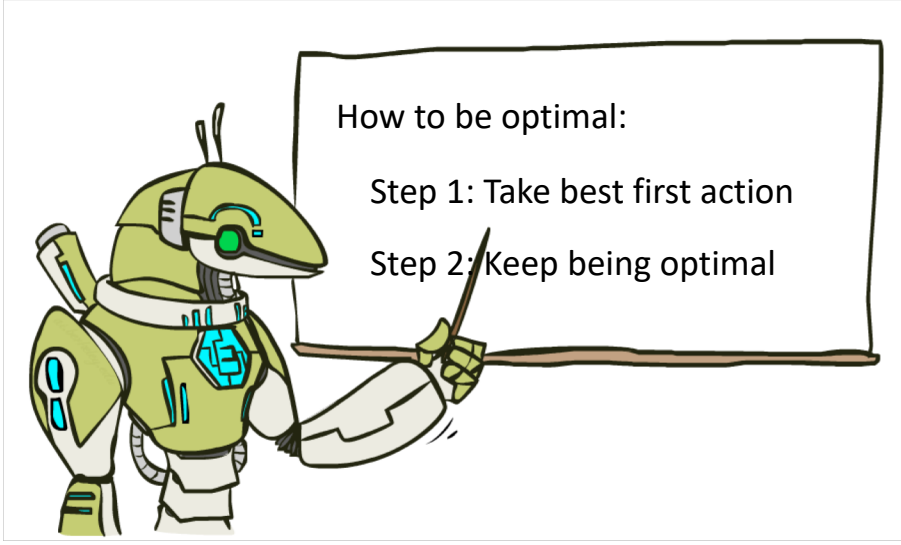
The value (utility) of the q-state (s,a):

$Q^*(s,a)$

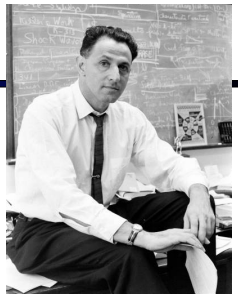
"expected utility of 1) starting in state ***s***
2) first taking action ***a***
3) acting *optimally* (ala π^*) forever after that"

Q*(s,a) = reward from executing a in s then ending in some s'
plus… discounted value of V*(s')

The Bellman Equations



The Bellman Equations

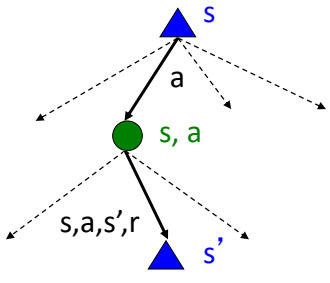

(1920-1984)

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

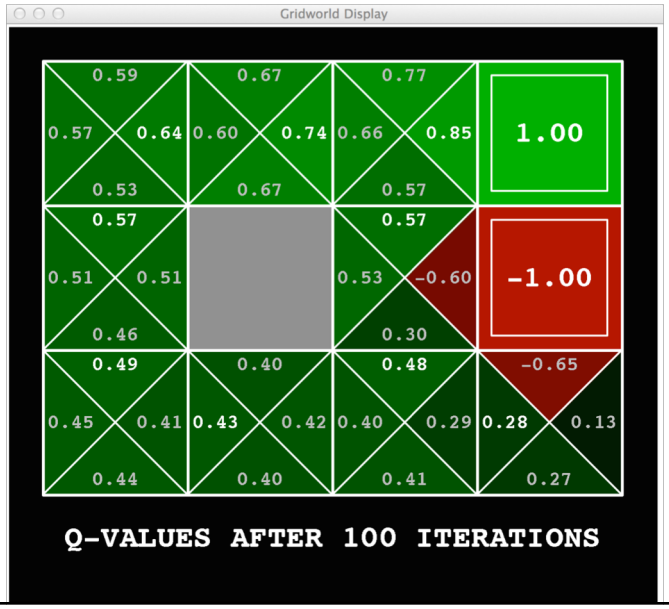
$$V^*(s) = \max_a Q^*(s,a)$$

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

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



Gridworld: Q*

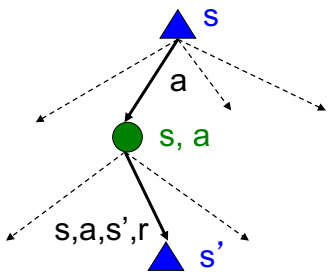


Gridworld Values V*

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

Values of States

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards



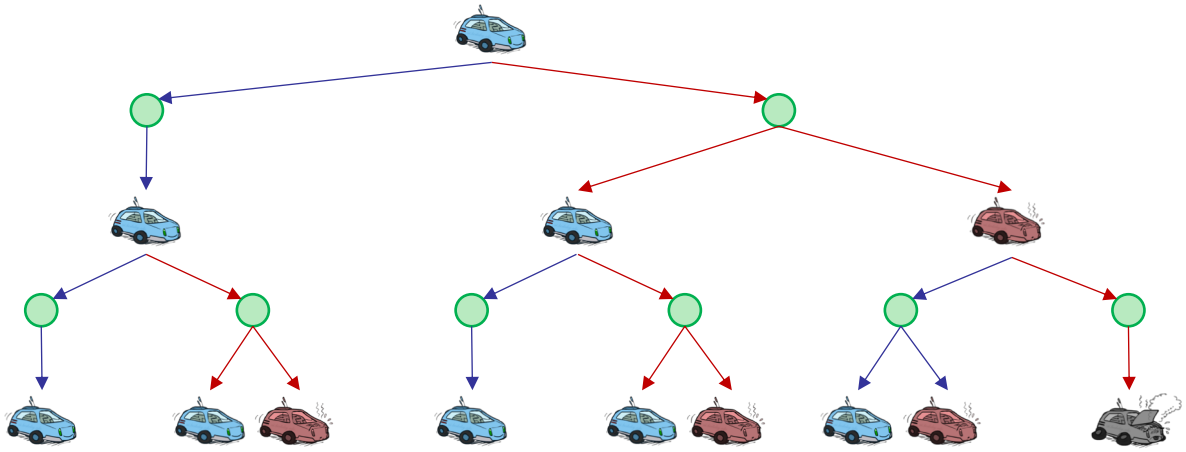
- Recursive definition of value:

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

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

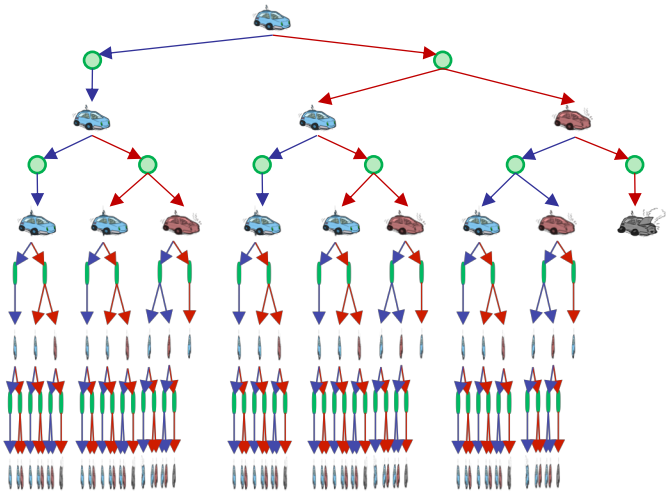
i.e. $$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

Racing Search Tree



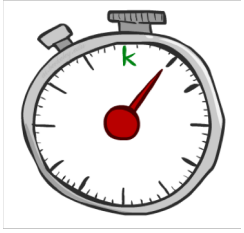
No End in Sight…

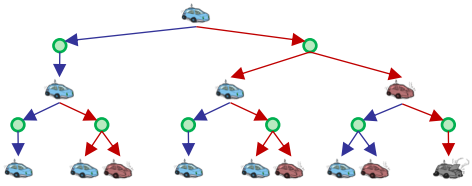
- Problem 1: Tree goes on forever
  - Rewards @ each step → **V changes**
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree *eventually* don't matter much (< ε) if $\gamma < 1$
- Problem 2: Too much repeated work
  - Idea: Only compute needed quantities once
  - Like **graph search** (*vs.* tree search)
  - Aka dynamic programming

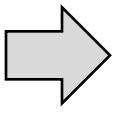


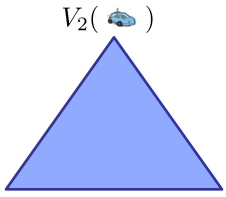
Time-Limited Values

- 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



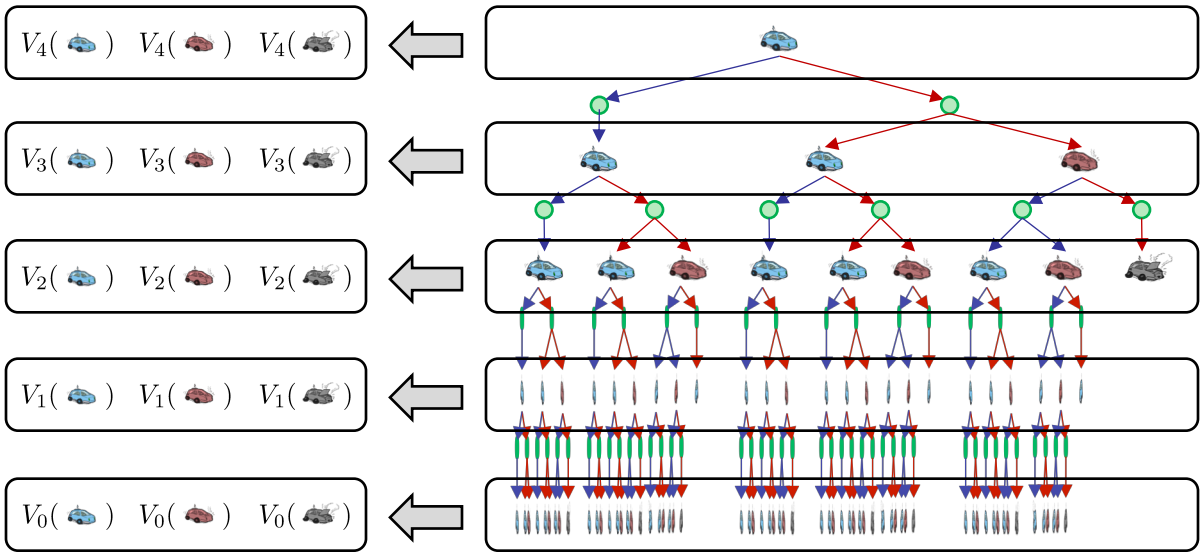




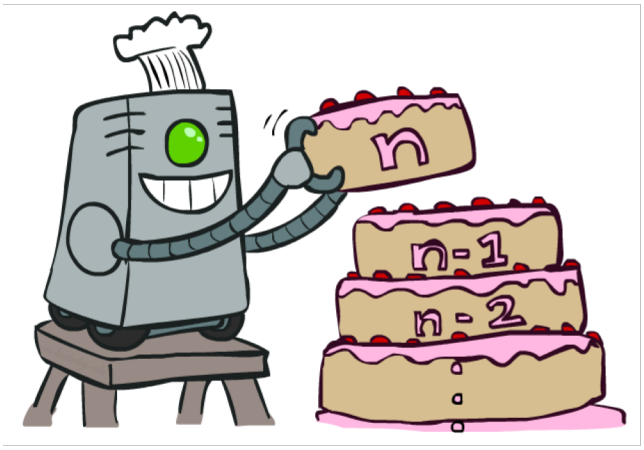


[Demo – time-limited values (L8D6)]

Time-Limited Values: Avoiding Redundant Computation



Value Iteration



Value Iteration

Called a "Bellman Backup"

- **Forall s, initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat**

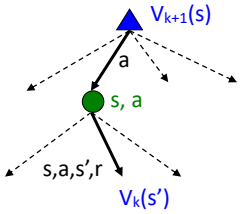
  K += 1

  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

  } do ∀s, a

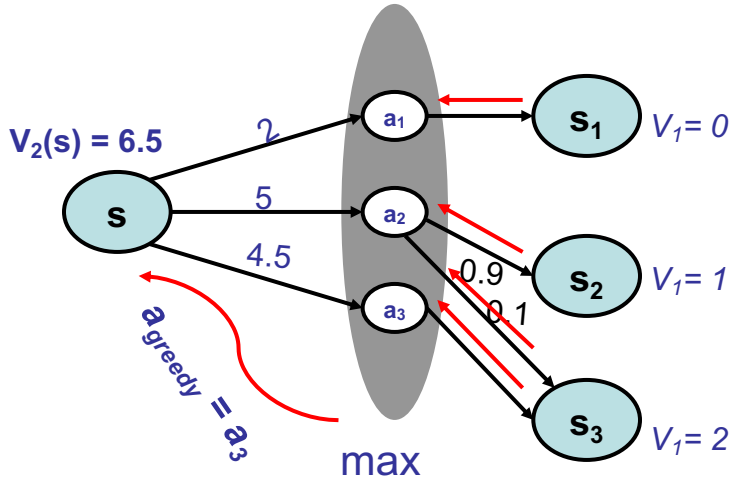
- **Repeat until $|V_{k+1}(s) - V_k(s)| < \varepsilon$, forall s** *"convergence"*



Successive approximation; dynamic programming

Example: Bellman Backup

Assume γ ~ 1



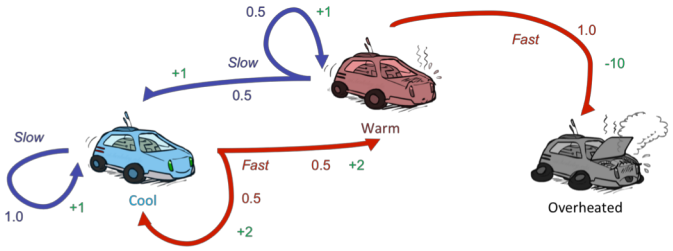
$Q_1(s,a_1) = 2 + \gamma\ 0$
~ 2

$Q_1(s,a_2) = 5 + \gamma\ 0.9 \sim 1$
$+ \gamma\ 0.1 \sim 2$
~ 6.1

$Q_1(s,a_3) = 4.5 + \gamma\ 2$
~ 6.5

Example: Value Iteration

Assume no discount (gamma=1) to keep math simple!



$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

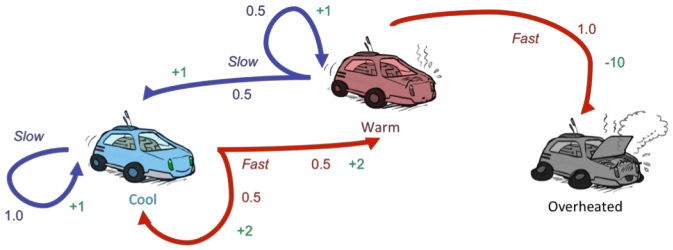
Example: Value Iteration

Assume no discount (gamma=1) to keep math simple!

V_0 | 0 | 0 | 0

V_1 |

V_2 |



$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

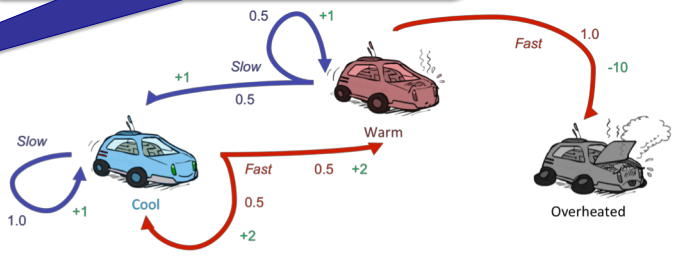
$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

Example: Value Iteration

math simple!

Q(,fast) =

Q(slow) =



V_0 0 0 0

$Q_1(s,a)$= 0

V_1 0

V_2

$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

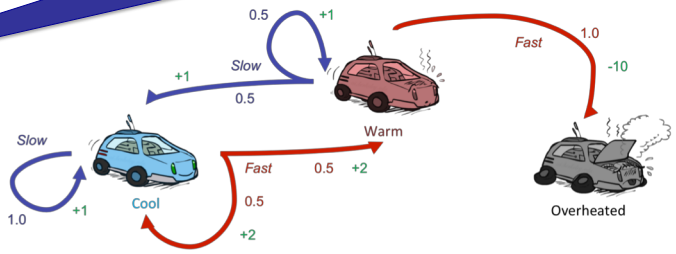
$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

Example: Value Iteration

math simple!

Q(,fast) = -10 + 0

Q(slow) = ½(1 + 0) + ½(1+0)



V_0 0 0 0

$Q_1(s,a)$= 1, -10 0

V_1 1 0

V_2

$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

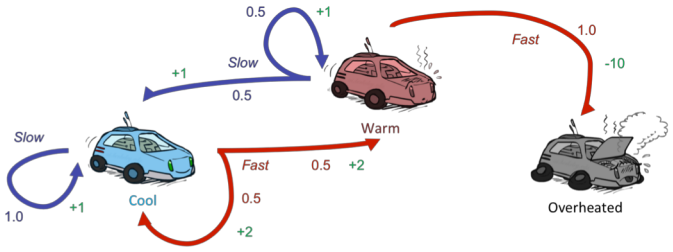
Ex…n

Q(🚙, fast) = ½(2 + 0) + ½(2 + 0)

Q(🚙, slow) = 1*(1 + 0)

…mma=1) to keep math simple!

V_0	0	0	0
$Q_1(s,a)$=	1, 2	1,-10	0
V_1	2	1	0
V_2			



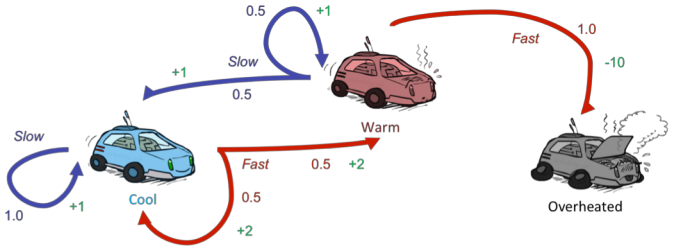
$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

Example: Value Iteration

Assume no discount (gamma=1) to keep math simple!

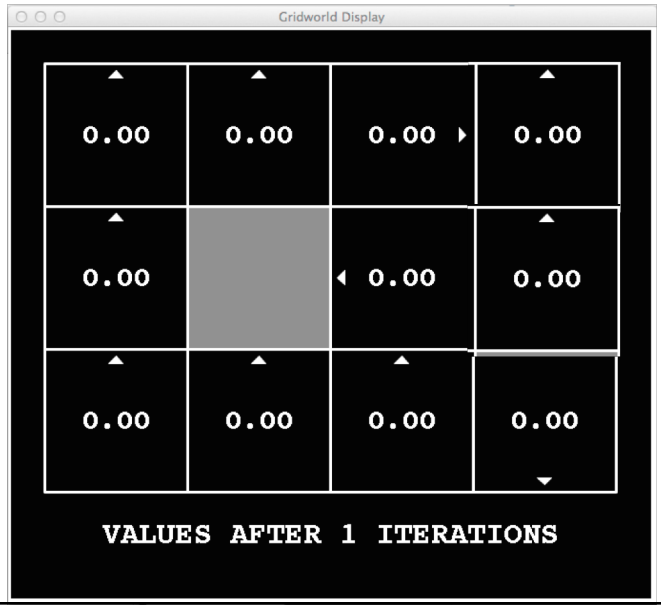
V_0	0	0	0
$Q_1(s,a)$=	1, 2	1,-10	0
V_1	2	1	0
$Q_2(s,a)$=	3,3.5	2.5,-10	0
V_2	3.5	2.5	0



$Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

$V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

k=0

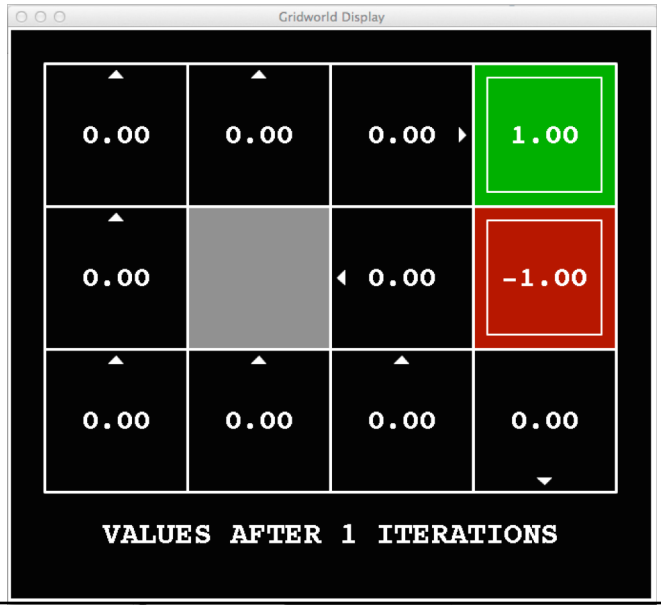


Noise = 0.2
Discount = 0.9
Living reward = 0

k=1

If agent is in 4,3, it only has one legal action: get jewel. It gets a reward and the game is over.
If agent is in the pit, it has only one legal action, die. It gets a penalty and the game is over.

Agent does NOT get a reward for moving INTO 4,3.



Noise = 0.2
Discount = 0.9
Living reward = 0

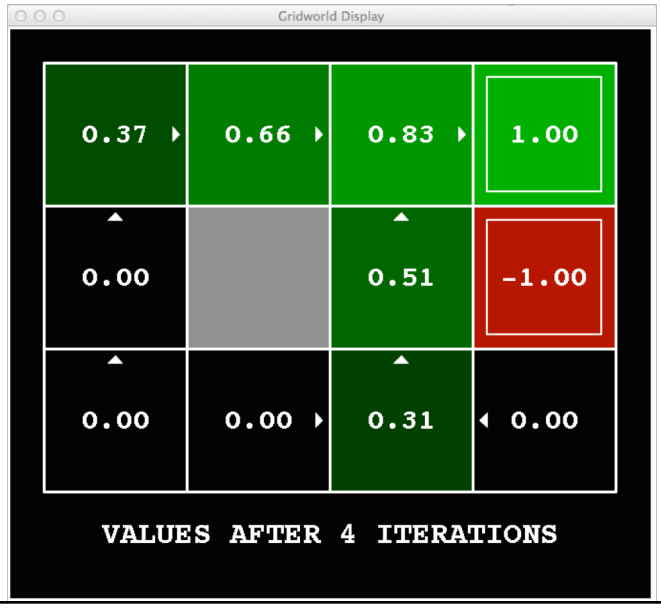
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

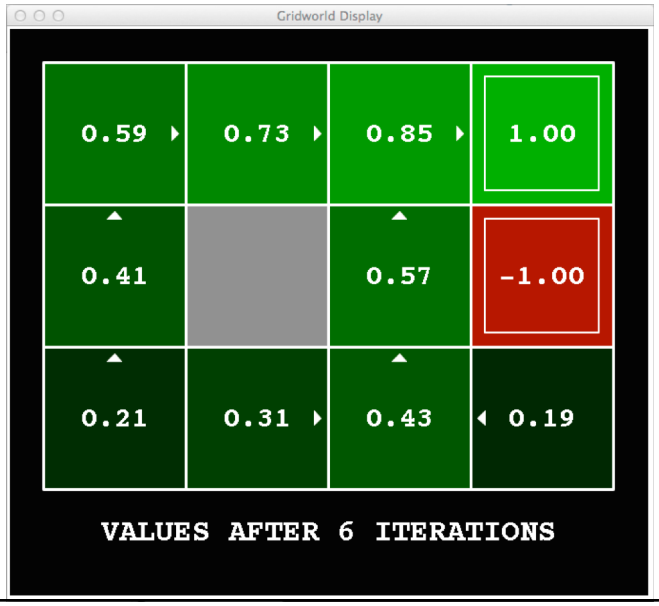


Noise = 0.2
Discount = 0.9
Living reward = 0

k=5

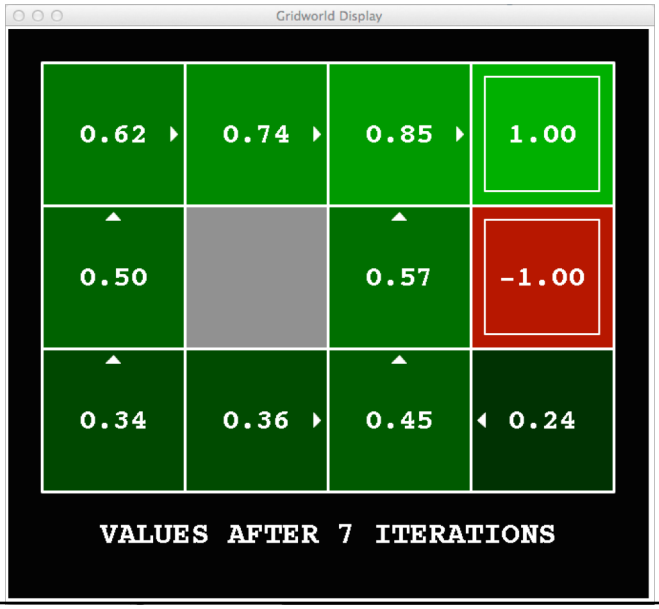
Noise = 0.2
Discount = 0.9
Living reward = 0

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

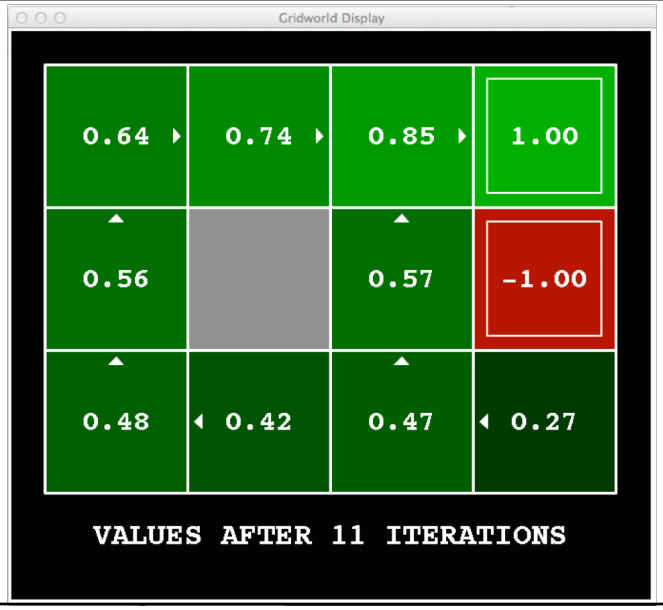
k=9

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

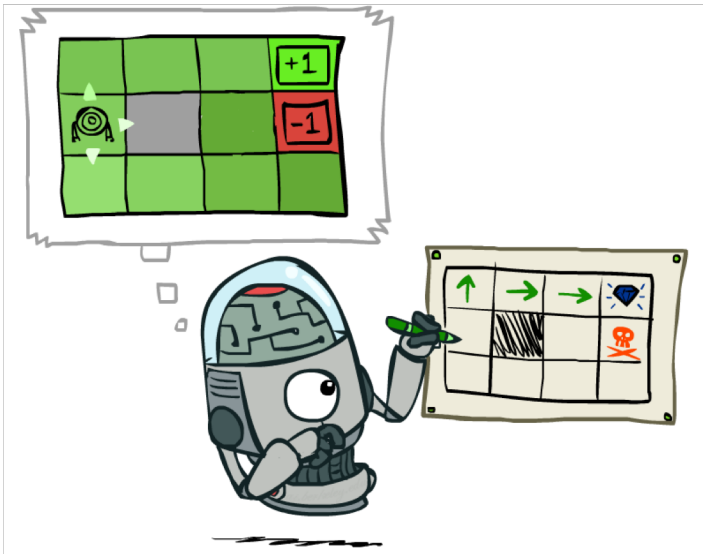
k=12

Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

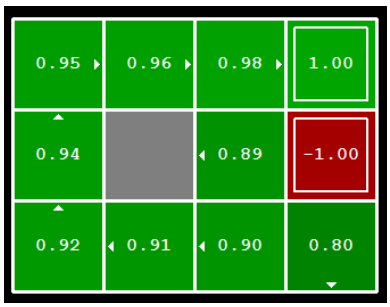
Noise = 0.2
Discount = 0.9
Living reward = 0

VI: Policy Extraction



Computing Actions from Values

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



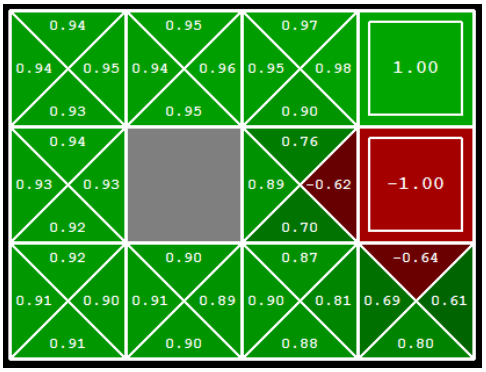
$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

Computing Actions from Q-Values

- Let's imagine we have the optimal q-values:
- How should we act?
  - Completely trivial to decide!

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



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

Value Iteration - Recap

- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat** *do Bellman backups*

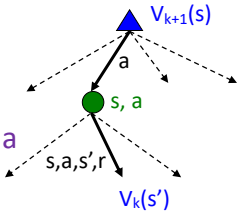
  K += 1

  Repeat for all states, s, and all actions, a:

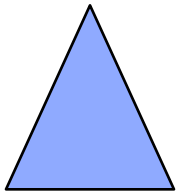
  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

  } do ∀s, a

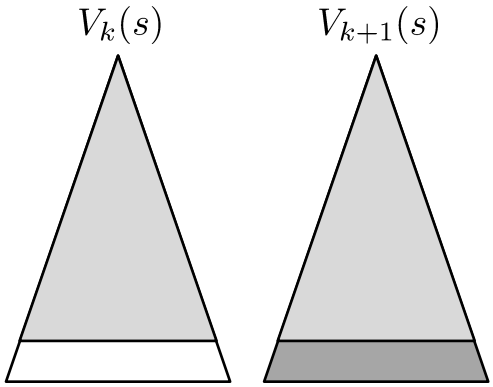


- **Until $|V_{k+1}(s) - V_k(s)| < \varepsilon$, forall s** *"convergence"*
- **Theorem: will converge to unique optimal values**



Convergence*

- How do we know the V_k vectors will 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 max difference happens if big reward at k+1 level
  - That last layer is at best all R_{MAX}
  - But everything is discounted by γ^k that far out
  - So V_k and V_{k+1} are at most γ^k max|R| different
  - So as k increases, the values converge



Value Iteration - Recap

- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat** *do Bellman backups*

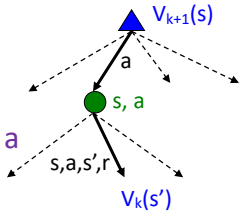
  K += 1

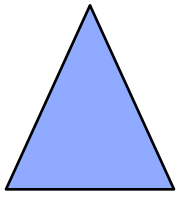
  Repeat for all states, s, and all actions, a:

  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

  } do ∀s, a





- **Until $|V_{k+1}(s) - V_k(s)| < \varepsilon$, forall s** *"convergence"*
- **Complexity of each iteration?**

Value Iteration - Recap

- **Forall s, Initialize $V_0(s) = 0$** *no time steps left means an expected reward of zero*
- **Repeat** *do Bellman backups*

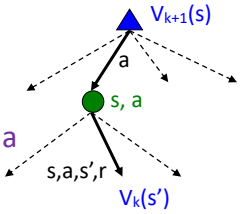
  K += 1

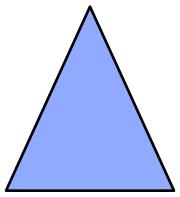
  Repeat for all states, s, and all actions, a:

  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$

  } do ∀s, a





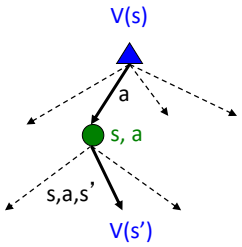
- **Until $|V_{k+1}(s) - V_k(s)| < \varepsilon$, forall s** *"convergence"*
- **Complexity of each iteration: $O(S^2A)$**
- **Number of iterations: poly(|S|, |A|, 1/(1-γ))**

Value Iteration as Successive Approximation

- Bellman equations ***characterize*** the optimal values:

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

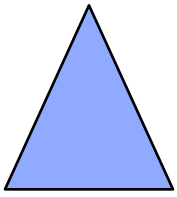
$$V^*(s) = \max_a Q^*(s,a)$$



- Value iteration ***computes*** them:

  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$



- Value iteration is just a ***fixed-point solution method***

  Computed using dynamic programming

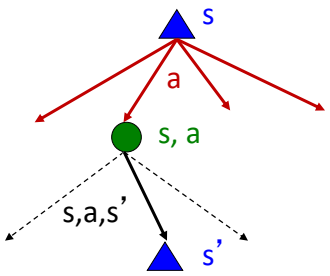
  … though the V_k vectors are also interpretable as time-limited values

Problems with Value Iteration

- Value iteration repeats the Bellman updates:

  $Q_{k+1}(s, a) = \Sigma_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$

  $V_{k+1}(s) = \text{Max}_a Q_{k+1}(s, a)$



- 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

Recap

- Bellman's Equations characterize the optimal policy

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

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

- Value iteration is an iterative method for finding the fixed point

Question 6 – Stutter Step MDP and Bellman Equations – 25 points

Consider the following special case of the general MDP formulation we studied in class. Instead of specifying an arbitrary transition distribution $T(s, a, s')$, the stutter step MDP has a function $T(s, a)$ that returns a next state s' deterministically. However, when the agent actually acts in the world, it often stutters. It only actually reaches s' half of the time, and it otherwise stays in s. The reward $R(s, a, s')$ remains as in the general case.

1. Write down a set of Bellman equations for the stutter step MDP in terms of $T(s, a)$, by defining $V^*(s)$, $Q^*(s, a)$ and $\pi^*(s)$. Be sure to include the discount γ. [25 pts]

VI → Asynchronous VI

- Is it essential to back up ***all*** states in each iteration?
  - No!
- States may be backed up
  - many times or not at all
  - in any order
- As long as no state gets starved…
  - convergence properties still hold!!

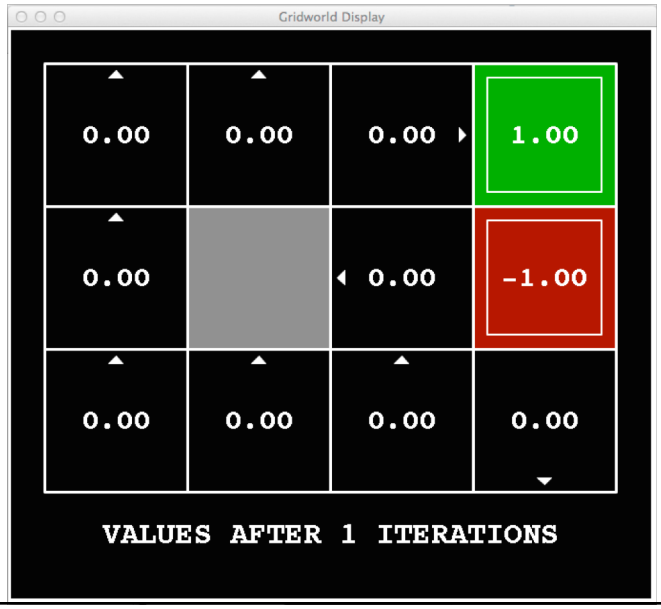
83

Prioritization of Bellman Backups

- Are all backups equally important?
- Can we avoid some backups?
- Can we schedule the backups more appropriately?

84

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

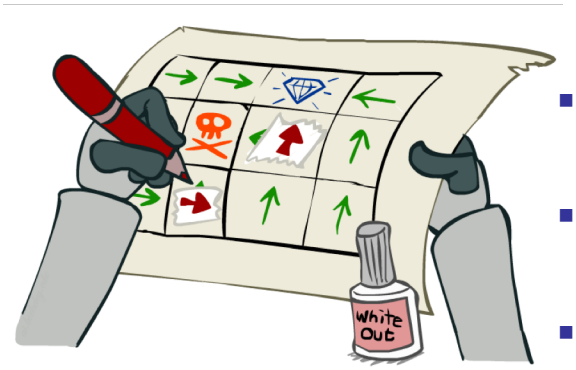
Noise = 0.2
Discount = 0.9
Living reward = 0

Asynch VI: Prioritized Sweeping

- Why backup a state if values of successors ***unchanged***?
- Prefer backing a state
  - whose successors had ***most*** change
- Priority Queue of (state, expected change in value ~ residual)
- Residual at *s* with respect to *V*
  - magnitude(Δ*V(s)*) after one Bellman backup at s

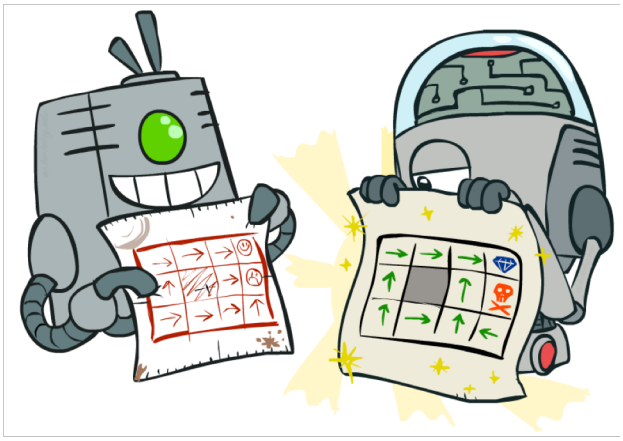
$$\text{Res}_V(s) = \left| V(s) - \max_{a \in A} \sum_{s' \in S} T(s,a,s')[R(s,a,s')+V(s')] \right|$$

Solving MDPs



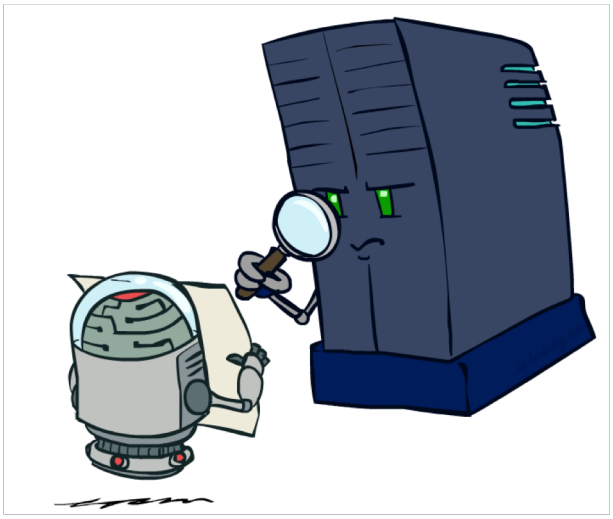
- Value Iteration
- Policy Iteration
- Heuristic Search Methods
- Real-Time Dynamic programming
- Reinforcement Learning

Policy Iteration



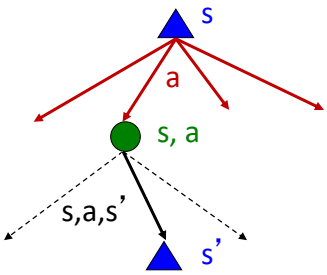
1. Policy Evaluation
2. Policy Improvement

Part 1 - Policy Evaluation

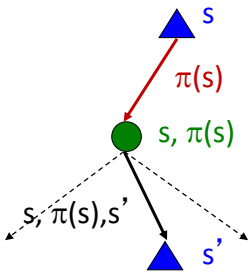


Fixed Policies

Do the optimal action



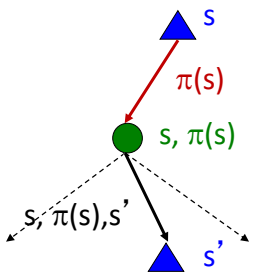
Do what π says to do



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

Computing Utilities for a Fixed Policy

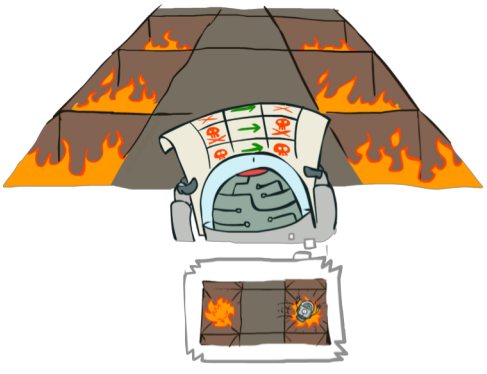
- A new basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy
- Define the utility of a state s, under a fixed policy π:
  $V^{\pi}(s)$ = expected total discounted rewards starting in s and following π
- Recursive relation (variation of Bellman equation):



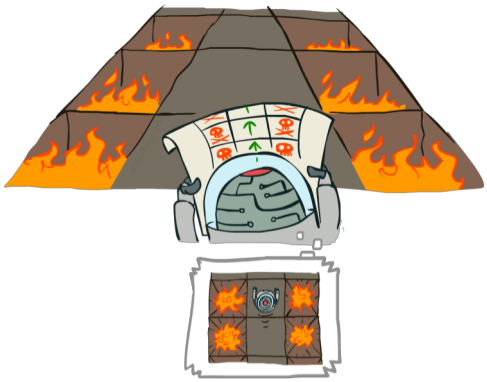
$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Example: Policy Evaluation

Always Go Right



Always Go Forward



Example: Policy Evaluation

Always Go Right

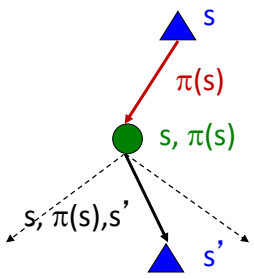
Always Go Forward

Iterative Policy Evaluation Algorithm

- How do we calculate the V's for a fixed policy π?
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

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

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

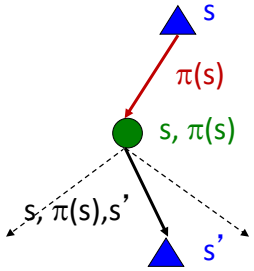


- Efficiency: $O(S^2)$ per iteration
  - Often converges in much smaller number of iterations compared to VI

Linear Policy Evaluation Algorithm

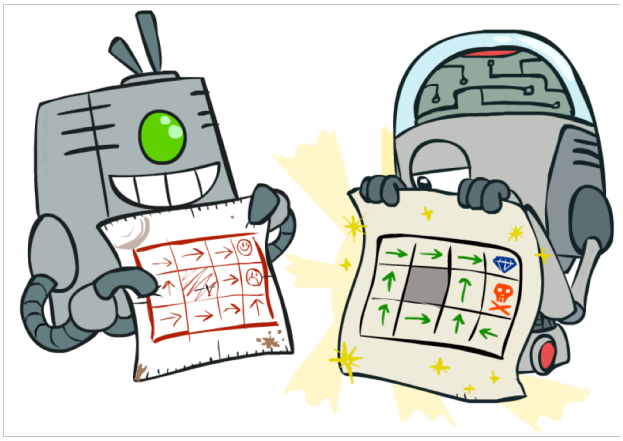
- Another way to calculate the V's for a fixed policy π?
- Idea 2: Without the maxes, the Bellman equations are just a linear system of equations

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



- Solve with Matlab (or your favorite linear system solver)
  - S equations, S unknowns = $O(S^3)$ and EXACT!
  - In large spaces, still too expensive

Policy Iteration



1. Policy Evaluation
2. Policy Improvement

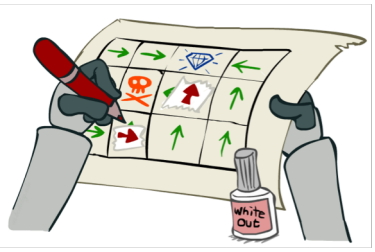
Policy Iteration

- Initialize π(s) to random actions
- Repeat
  - Step 1: Policy evaluation: calculate utilities of π at each s (*e.g.*, using a nested loop)
  - Step 2: Policy improvement: update policy using one-step look-ahead

    For each s, what's the best action to execute, ***assuming agent then follows π?***

    Let π'(s) = this best action.

    π = π'
- Until policy doesn't change



Policy Iteration Details

- Initialize π(s) to random actions
- Repeat
  - Step 1: Policy evaluation:
    - Initialize k=0; Forall s, $V_0^\pi(s) = 0$
    - Repeat until V^π converges
      - For each state s, $V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$
      - Increment k
  - Step 2: Policy improvement:
    - For each state, s, $\pi'(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^\pi(s')\right]$
    - If π == π' then it's optimal; return it.
    - Else set π := π' and loop.

Example

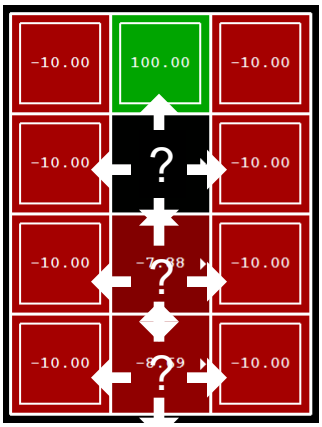
Initialize π_0 to "always go right"

Perform policy evaluation

Perform policy improvement
Iterate through states

Has policy changed?

Yes! i += 1



Example

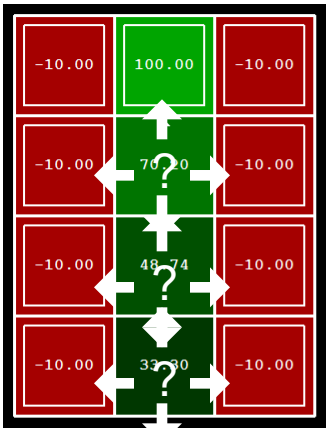
π_1 says "always go up"

Perform policy evaluation

Perform policy improvement
Iterate through states

Has policy changed?

No! We have the optimal policy



Policy Iteration Properties

- Can we view PI as search?
  - Space of …?
  - Algorithm?
- Policy iteration finds the optimal policy, guaranteed (assuming exact evaluation)!
  - Why does hill-climbing yield optimum?!?
- Often converges (much) faster than VI

Modified Policy Iteration [van Nunen 76]

- initialize π_0 as a random [proper] policy
- Repeat

  Approximate Policy Evaluation: Compute $V^{\pi_{n-1}}$

  by running only few iterations of iterative policy eval.

  Policy Improvement: Construct π_n greedy wrt $V^{\pi_{n-1}}$
- Until convergence
- return π_n

110

Comparison

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
  - What is the space being searched?
- In policy iteration:
  - We do fewer iterations
  - Each one is slower (must update all V^{π} and then choose new best π)
  - What is the space being searched?
- Both are dynamic programs for planning in MDPs

Comparison II

- Changing the search space.
- Policy Iteration
  - Search over policies
  - Compute the resulting value
- Value Iteration
  - Search over values
  - Compute the resulting policy

Summary So Far: MDP Algorithms

- So you want to….
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)
- These all look the same!
  - They all use variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ
    - whether we plug in a fixed policy or max over actions
    - Whether we search finite (policy) space or infinite (R valued value function) space
- Next time we'll see some other methods

What's Next Part II? Reinforcement Learning!

- So far we've assumed agent knows T(s,a,s') and R(s,a,s')
- Often one doesn't know them, must interact to learn them!
  - PS4 (after midterm) will cover this