Artificial Intelligence

PART 6.4 : Decisions Theory

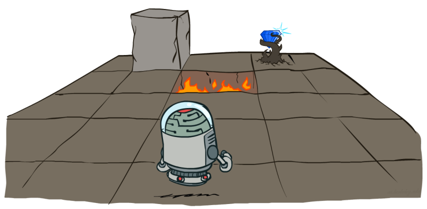
PART 6.4 : Markov Decision Processes

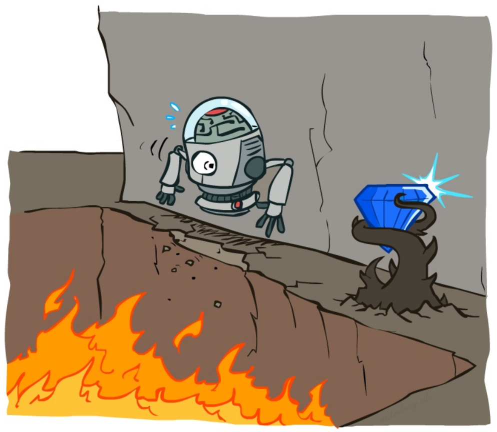
Dr. Chandra Prakash
Assistant Professor
Department of Computer Science and Engineering

(Slides adapted from StuartJ. Russell, B Ravindran, Mausam, Dan Klein and Pieter Abbeel, Partha P Chakrabarti, Saikishor Jangiti

Module 6: Reasoning under Uncertainty

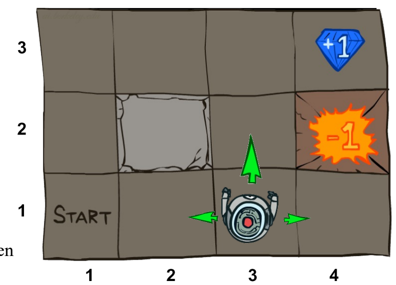
Non-Deterministic Search





Example: Grid World

- A maze-like problem
  - The agent lives in a grid (11)
  - Walls block the agent's path (2,2)
- 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



Decision Theory

- is the study of agent’s choices
- lays out principles for how an agent arrives at an optimal choice
  - Good decisions may occasionally have unexpected bad outcomes
    - it is still a good decision if made properly
  - Bad decisions may occasionally have good outcomes if you are lucky
    - it is still a bad decision

Steps in Decision Theory

1. List the possible actions (actions/decisions)
2. Identify the possible outcomes
3. List the payoff or profit or reward
4. Select one of the decision theory models
5. Apply the model and make your decision

Probabilistic Uncertainty

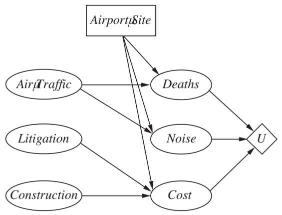
- Decision makers know the probability of occurrence for each possible outcome
  - Attempt to maximize the expected reward
- Criteria for decision models in this environment:
  - Maximization of expected reward
  - Minimization of expected regret
    - Minimize expected regret = maximizing expected reward!

Utility Function

- Utility
  - is a function that maps from states to real numbers.
- Utility of money
  - expected monetary value

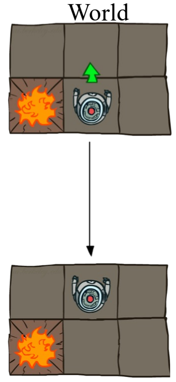
Decision Network

- general mechanism for making rational decisions
- also known as influence diagram
- Decision networks combine Bayesian networks with additional node type of actions and utilites.
- Representating a decision problem with a decision network :
  - A simple decision network for the airport-siting problem.
    - chance node
    - decision nodes
    - utility nodes

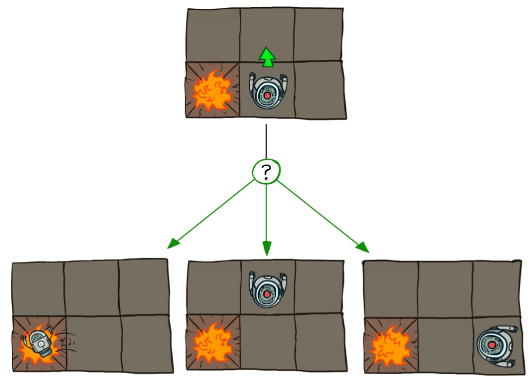


Grid World Actions

Deterministic Grid World

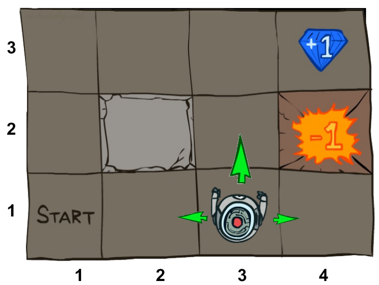


Stochastic Grid World



Markov Decision Processes (MDP)

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



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

Video of Demo Gridworld Manual Intro

What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means action outcomes depend only on the current state

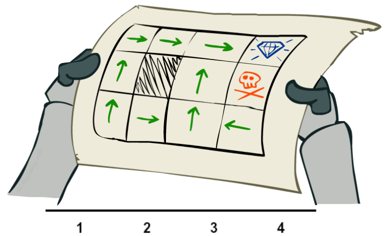
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$
$$=$$
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

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

Andrey Markov (1856-1922)

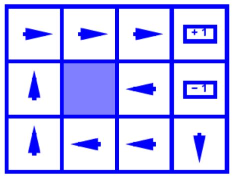
Policies

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

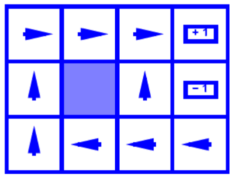


Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

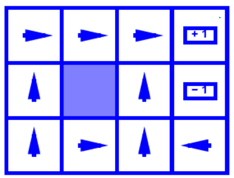
Optimal Policies



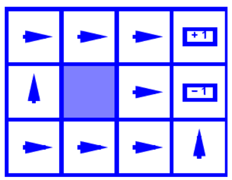
R(s) = -0.01



R(s) = -0.03



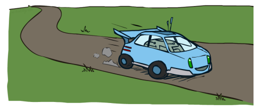
R(s) = -0.4

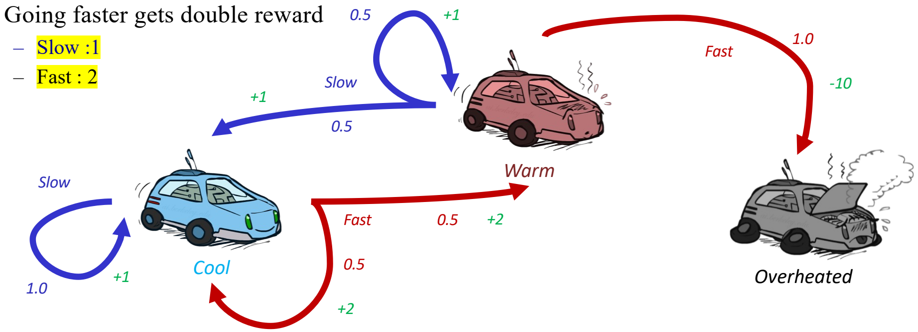


R(s) = -2.0

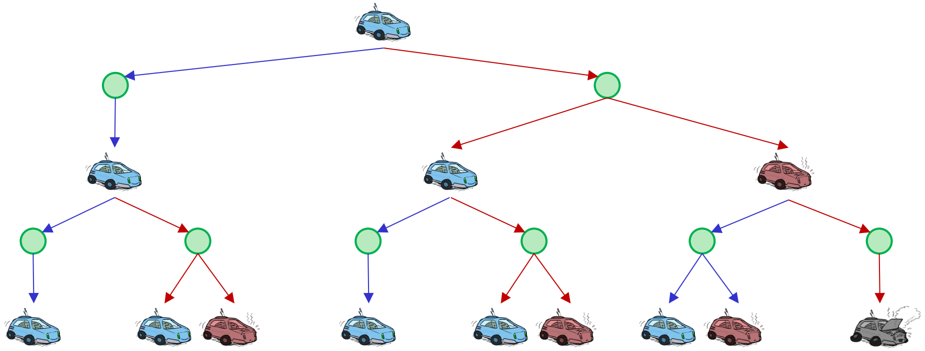
Example: Racing

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward
  - Slow :1
  - Fast : 2



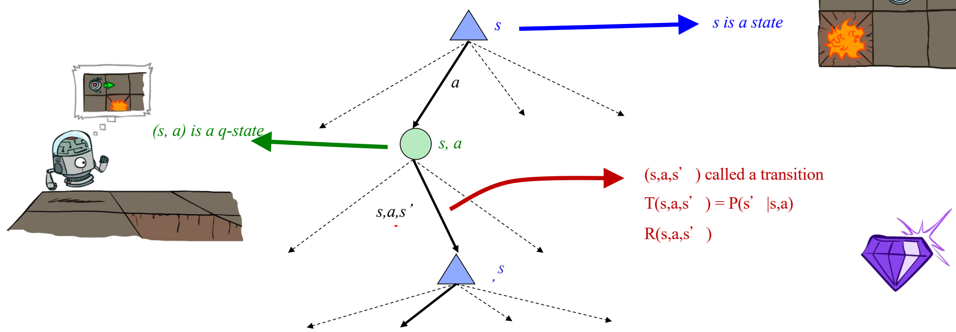


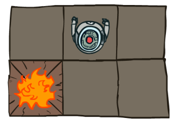
Racing Search Tree



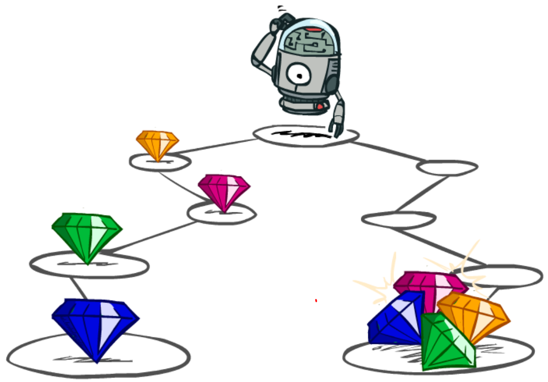
MDP Search Trees

- Each MDP state projects an expectimax-like search tree





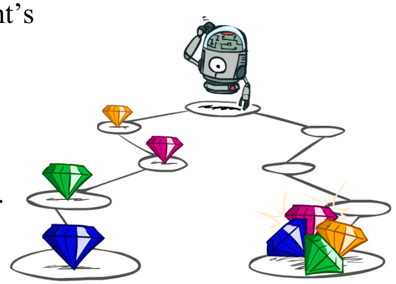
Utilities of Sequences



Utilities of Sequences

- Utilities are functions from outcomes (states of the world) to real numbers that describe an agent’s preferences
- Where do utilities come from?
  - In a game, may be simple (+1/-1)
  - Utilities summarize the agent’s goals
- What preferences should an agent have over reward sequences?

  [1, 2, 3] or [3, 2, 1]
- More or less? [0, 0, 1] or [1, 0, 0]
- Now or later?



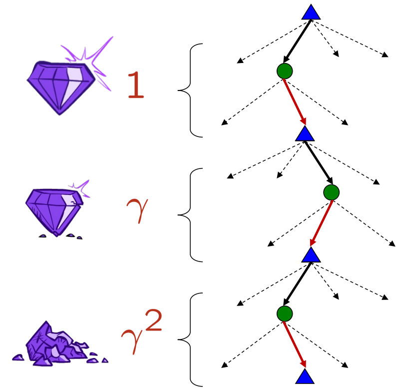
Discounting

- 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	γ	γ^2
Worth Now	*Worth Next Step*	*Worth In Two Steps*

Discounting

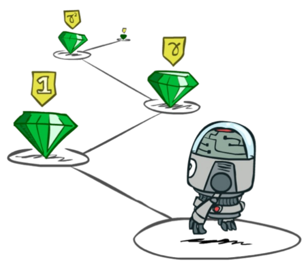
- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([3,2,1]) ???
  - U([1,2,3])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



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

Quiz: Discounting

- Given:

10				1
a	b	c	d	e

  - Actions: East, West, and Exit (only available in exit states a, e)
  - 0 rewards in non terminal states
  - Transitions: deterministic
- Quiz 1: For $\gamma = 1$, what is the optimal policy?
  - It will go West always

10				1
a	b	c	d	e

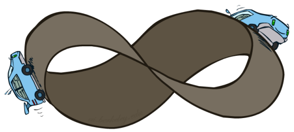
- Quiz 2: For $\gamma = 0.1$, what is the optimal policy?
  - if at state c : $0 + \gamma * 1 = 0.1$
  - if at state d : $0 + \gamma * 0 + \gamma^2 * 0 + \gamma^3 * 10 = 0.01$

10				1

- Quiz 3: For which γ are West and East equally good when in state d?
  - $1/\sqrt{10}$

Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)



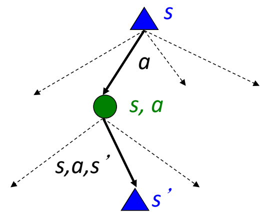
  - Discounting: use $0 < \gamma < 1$

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

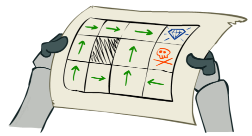
    - Smaller γ means smaller "horizon" – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)

Recap: Defining MDPs

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



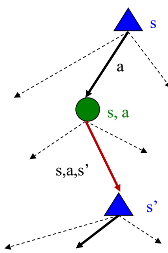
- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards
  - Values = Expected future utility from a state (max node)
  - Q-Values = Expected future utility from a q-state (chance node)



- Solving MDP

Optimal Quantities

- The value (utility) of a state s:
  $V^*(s)$ = expected utility starting in s and acting optimally
- The value (utility) of a q-state (s,a):
  $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally
- The optimal policy:
  $\pi^*(s)$ = optimal action from state s
  = arg Max Q*(s,a)

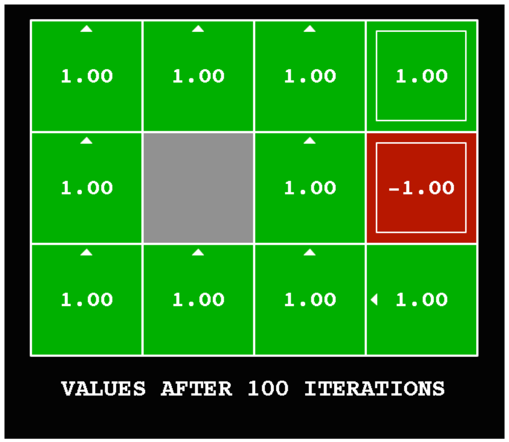


s is a state

(s, a) is a q-state

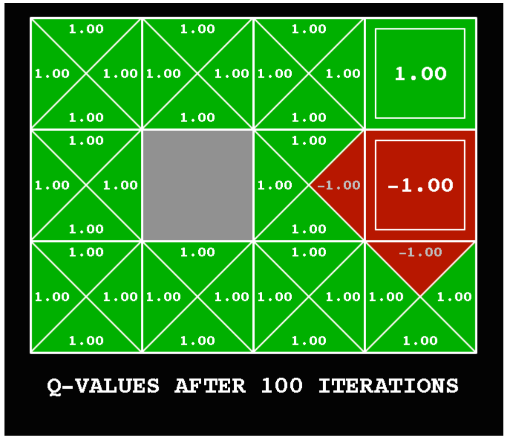
(s,a,s') is a transition

Snapshot of Demo – Gridworld V Values



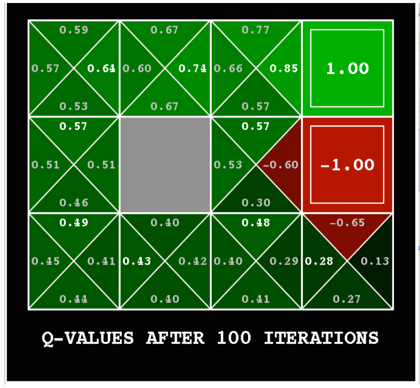
Noise = 0
Discount = 1
Living reward = 0

Snapshot of Demo – Gridworld Q Values



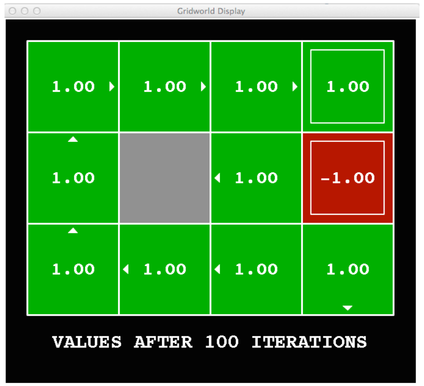
Noise = 0
Discount = 1
Living reward = 0

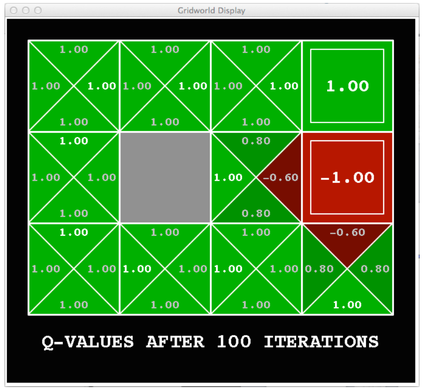
Snapshot of Demo – Gridworld V and Q Values



Noise = 0.2
Discount = 0.9
Living reward = 0

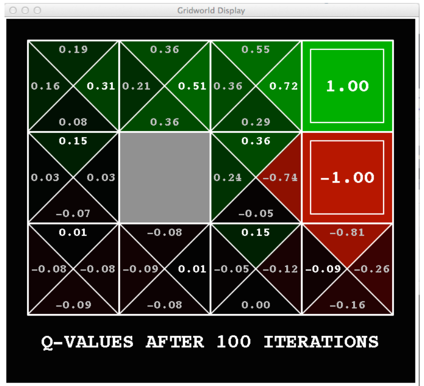
Snapshot of Demo – Gridworld V and Q Values





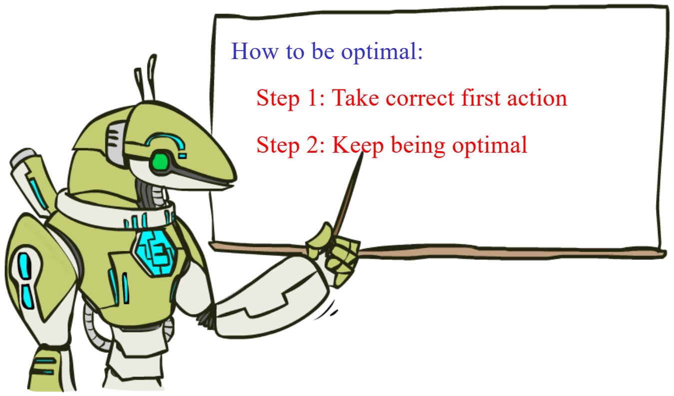
Noise = 0.2
Discount = 1
Living reward = 0

Snapshot of Demo – Gridworld V and Q Values



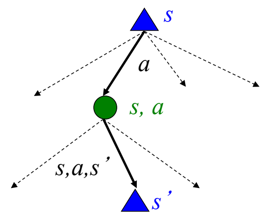
Noise = 0.2
Discount = 0.9
Living reward = -0.1

The Bellman Equations



Values of States

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!



- Recursive definition of value:

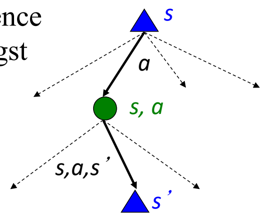
$$V^*(s) = \max_a Q^*(s,a)$$

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

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

The Bellman Equations

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



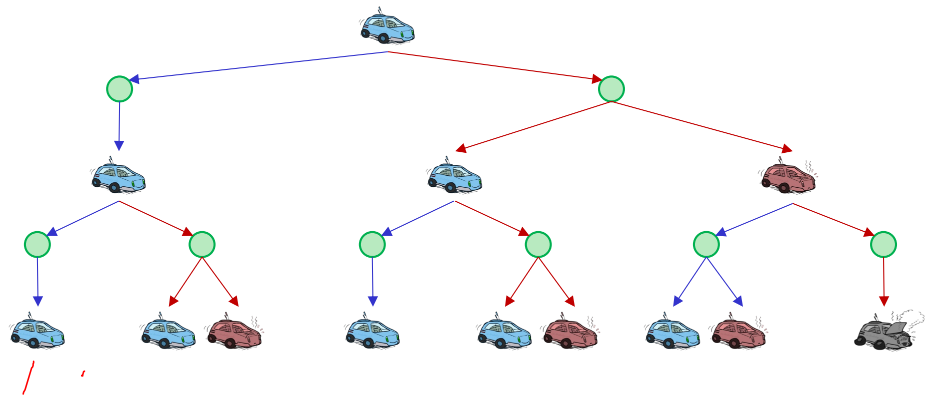
$$V^*(s) = \max_a Q^*(s,a)$$

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

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

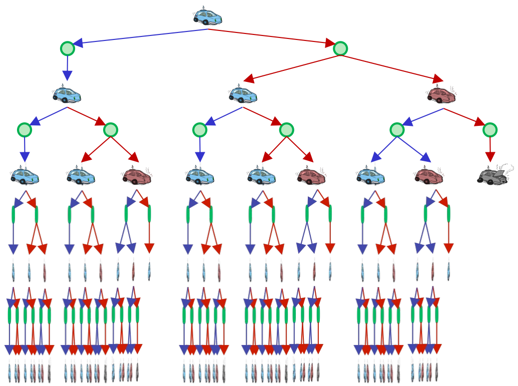
- These are the Bellman equations, and they characterize optimal values in a way we'll use over and over

Racing Search Tree



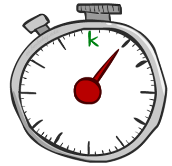
Racing Search Tree

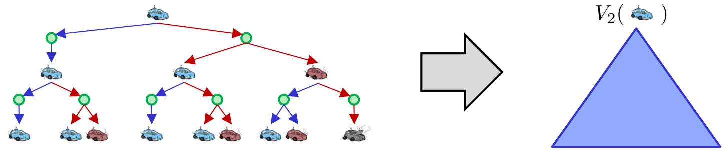
- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter if $\gamma < 1$



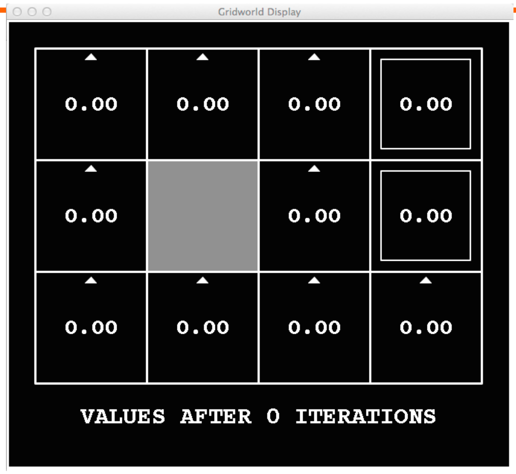
Time-Limited Values

- 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



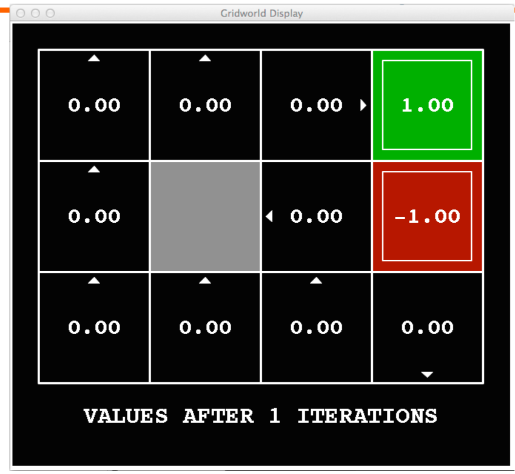


k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1

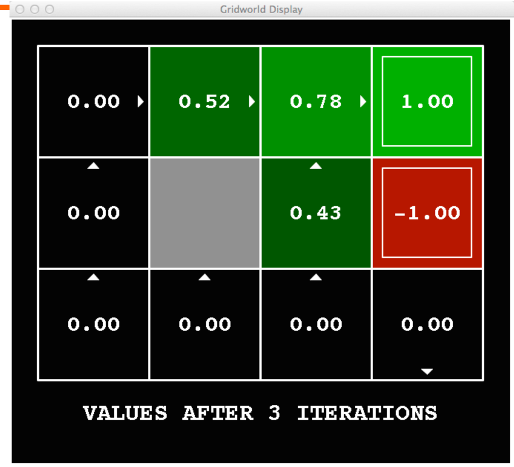


Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

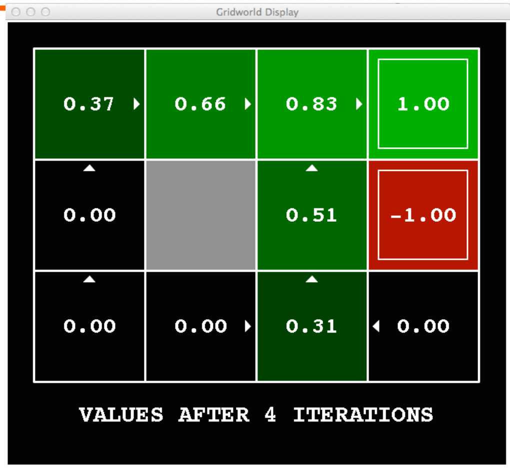
Noise = 0.2
Discount = 0.9
Living reward = 0

k=3



Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

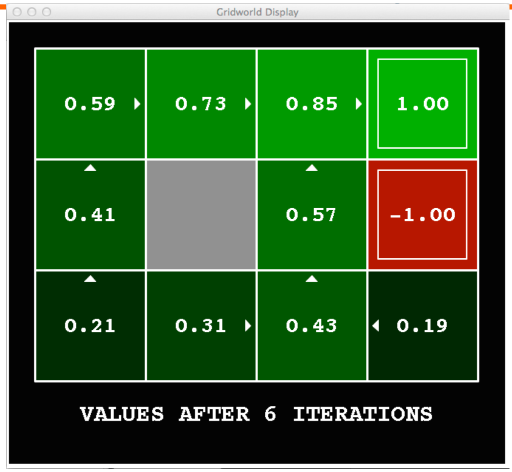


Noise = 0.2
Discount = 0.9
Living reward = 0

k=5

Noise = 0.2
Discount = 0.9
Living reward = 0

k=6

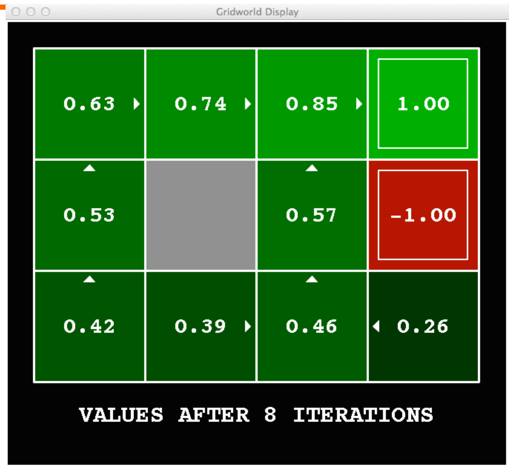


Noise = 0.2
Discount = 0.9
Living reward = 0

k=7

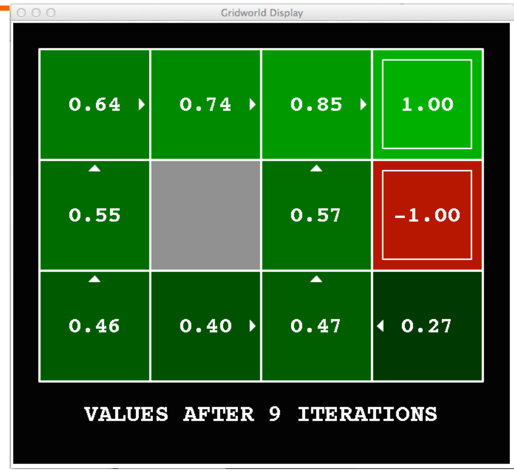
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



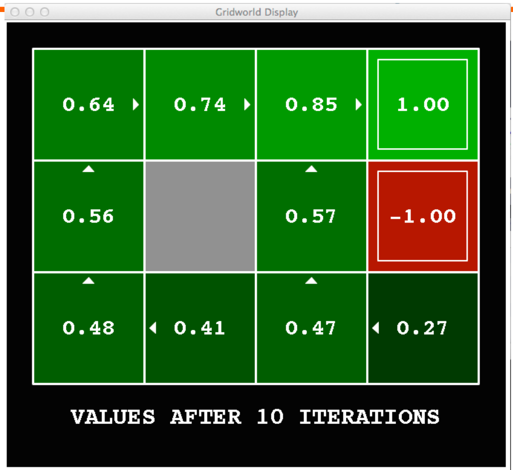
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



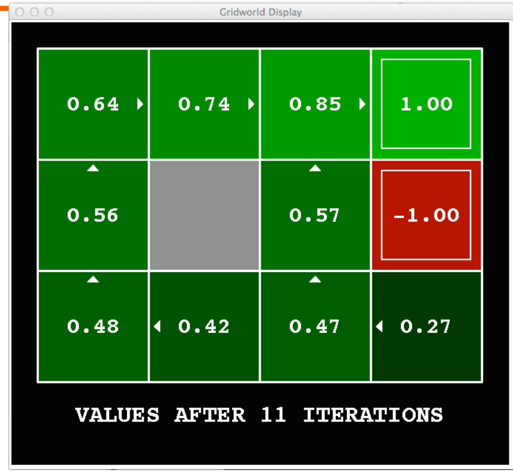
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



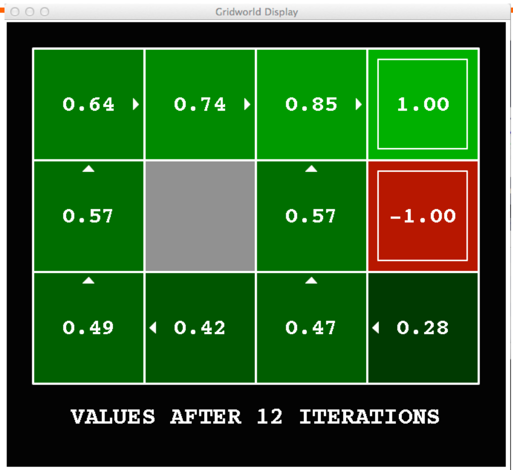
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

k=12

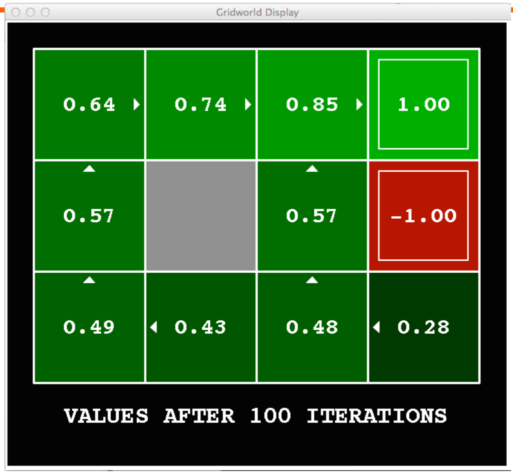


Noise = 0.2
Discount = 0.9
Living reward = 0

k=24

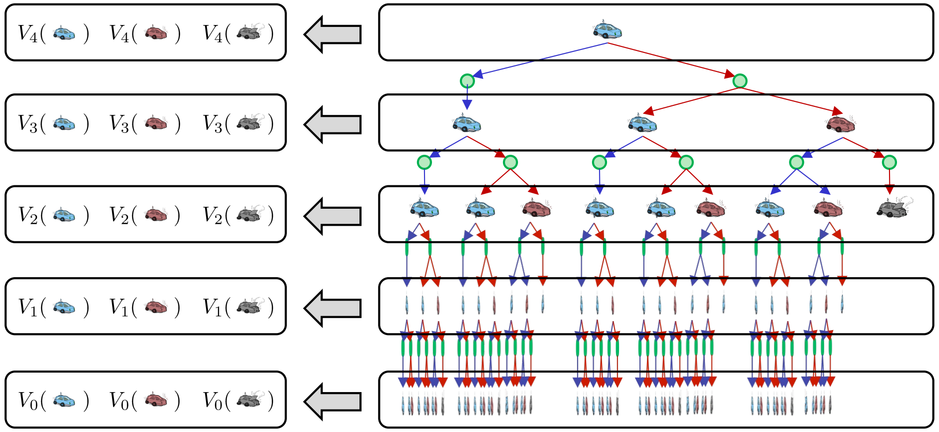
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

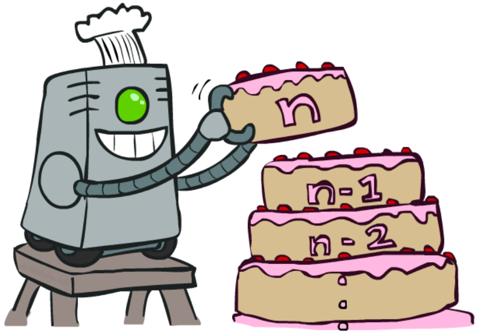


Noise = 0.2
Discount = 0.9
Living reward = 0

Computing Time-Limited Values



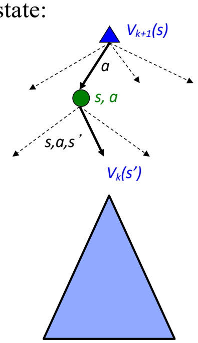
Value Iteration



Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

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



- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

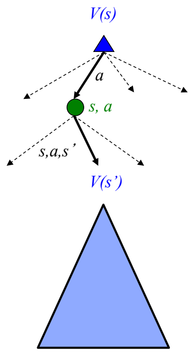
Value Iteration

- Bellman equations characterize the optimal values:

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

- Value iteration computes them:

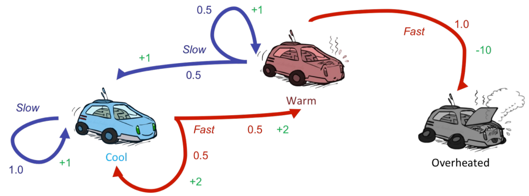
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$



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

Example: Value Iteration

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

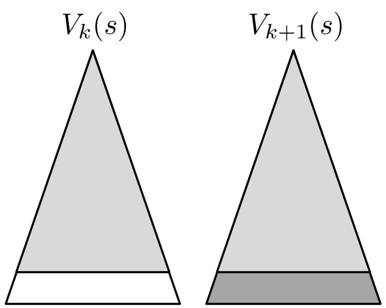


Assume no discount!

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

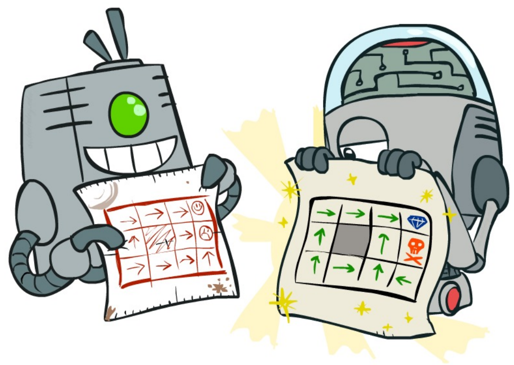
Convergence*

- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
  - That last layer is at best all R_{MAX}
  - It is at worst R_{MIN}
  - But everything is discounted by γ^k that far out
  - So V_k and V_{k+1} are at most γ^k max|R| different
  - So as k increases, the values converge

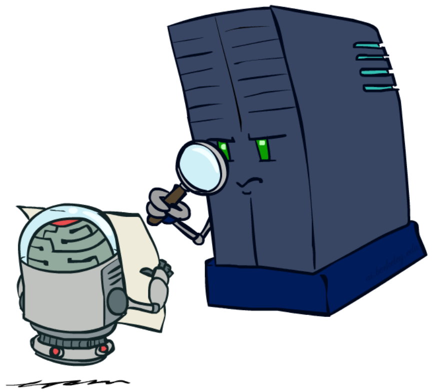


Policy Methods

- Different ways to solve same problem

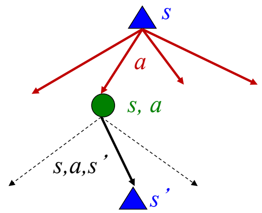


Policy Evaluation

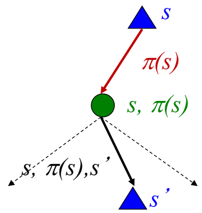


Fixed Policies

Do the optimal action

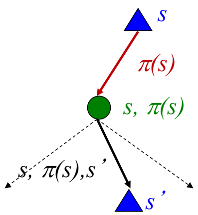


Do what π says to do



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

Utilities for a Fixed Policy

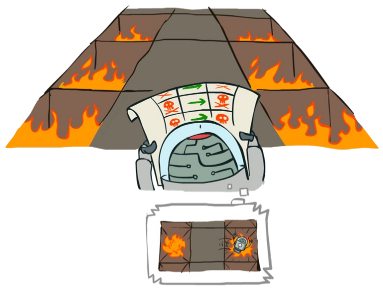


- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy
- Define the utility of a state s, under a fixed policy π:
  - $V^\pi(s)$ = expected total discounted rewards starting in s and following π
- Recursive relation (one-step look-ahead / Bellman equation):

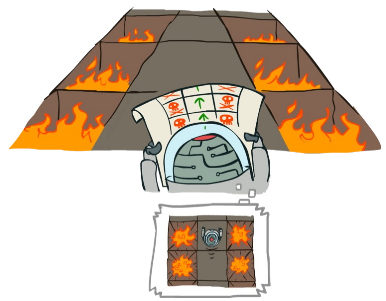
$$\boxed{V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}(s')]}$$

Example: Policy Evaluation

Always Go Right



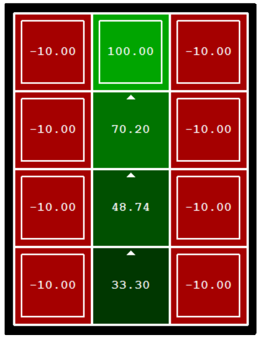
Always Go Forward



Example: Policy Evaluation

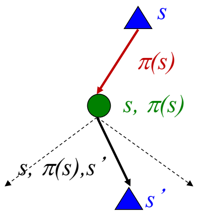
Always Go Right

Always Go Forward



Policy Evaluation

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



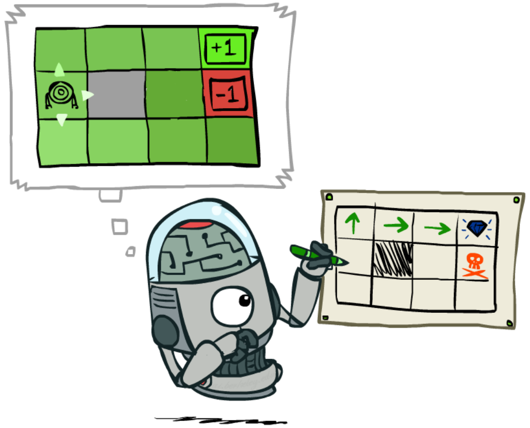
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

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

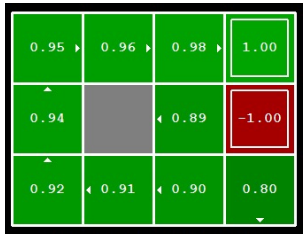
$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

- Efficiency: $O(S^2)$ per iteration
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with python (or your favorite linear system solver)

Policy Extraction



Computing Actions from Values



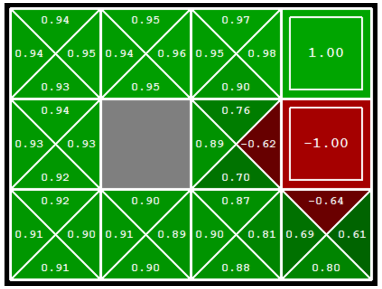
- Let's imagine we have the optimal values $V^*(s)$
- How should we act?
  - It's not obvious!
- We need to do a mini-expectimax (one step)

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

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

Computing Actions from Q-Values

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

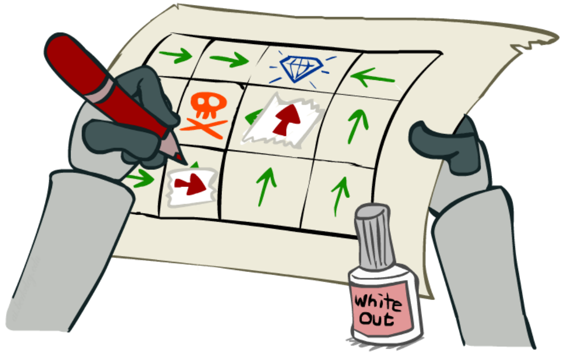


- How should we act?
  - Completely trivial to decide!

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

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

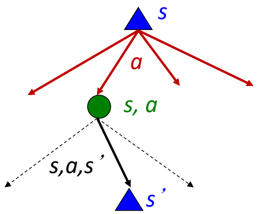
Policy Iteration



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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



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

Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges
- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

Policy Iteration

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

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

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

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

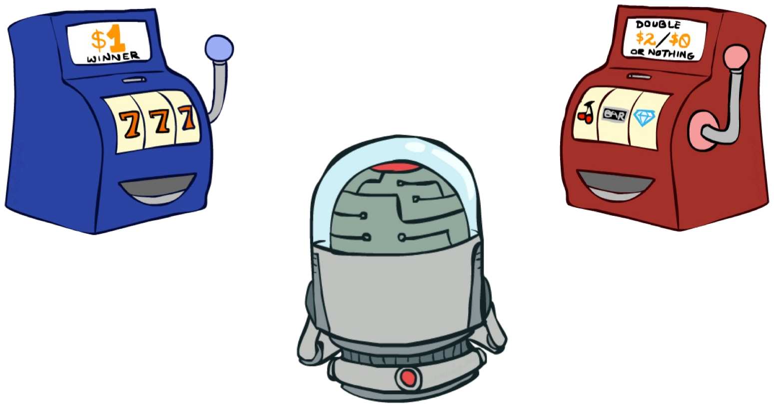
Comparison

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)
- Both are dynamic programs for solving MDPs

Summary: MDP Algorithms

- So you want to….
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)
- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step look ahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions

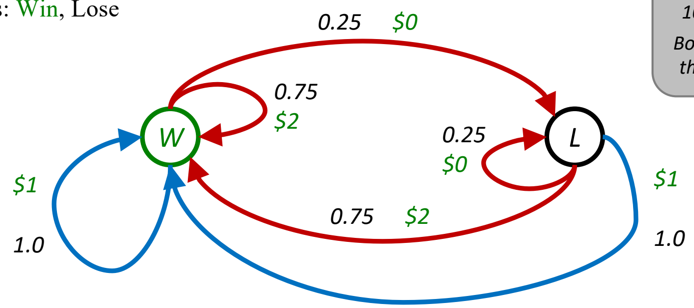
Double Bandits



Double-Bandit MDP

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

No discount
100 time steps
Both states have the same value

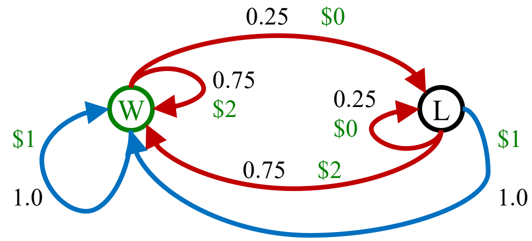


Offline Planning

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

No discount
100 time steps
Both states have the same value

	Value
Play Red	150
Play Blue	100



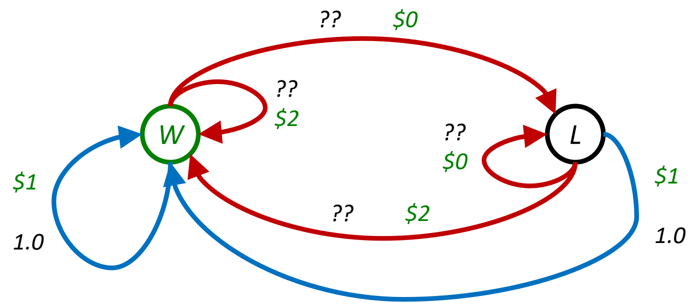
Let's Play!

$2 $2 $0 $2 $2

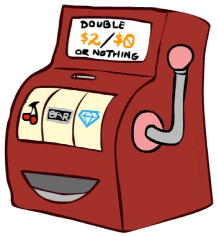
$2 $2 $0 $0 $0

Online Planning

- Rules changed! Red's win chance is different.



Let's Play!



\$0 \$0 \$0 \$2 \$0

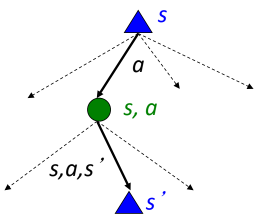
\$2 \$0 \$0 \$0 \$0

What Just Happened?

- That wasn't planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn't solve it with just computation
  - You needed to actually act to figure it out
- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP

Recap: MDPs

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



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

Optimal Quantities

- The value (utility) of a state s:

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

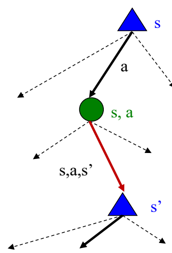
- The value (utility) of a q-state (s,a):

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

- The optimal policy:

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

  = argmax $Q^*(s,a)$

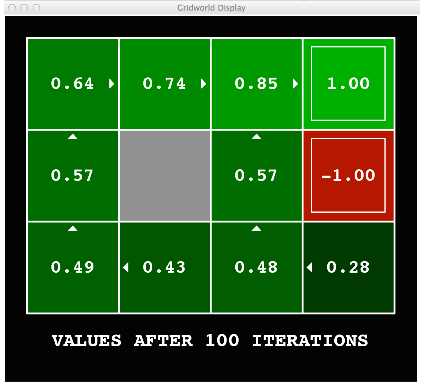


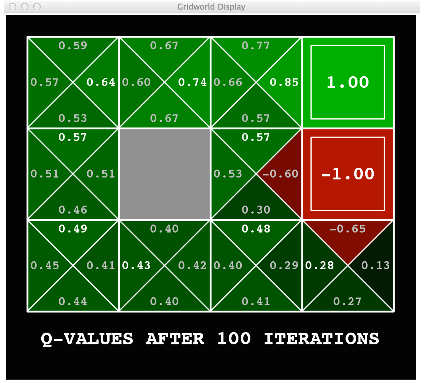
s is a state

(s, a) is a q-state

(s,a,s') is a transition

Gridworld Values V*





Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges
- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

Next :

- Module 7: Reinforcement Learning
  - PART 7.1 : Introduction to Learning
  - PART 7.2 : Types of ML
  - PART 7.3 : Reinforcement Learning
    - Key Concepts
  - PART 7.4 : Model based and model free learning
  - PART 7.5 : TD and Q Learning

References

- ***Artificial Intelligence** by Elaine Rich & Kevin Knight*, Third Ed, Tata McGraw Hill
- ***Artificial Intelligence and Expert System** by Patterson*
- http://www.cs.rmit.edu.au/AI-Search/Product/
- http://aima.cs.berkeley.edu/demos.html (for more demos)
- ***Artificial Intelligence and Expert System** by Patterson*
- Slides adapted from CS188 Instructor: Anca Dragan, University of California, Berkeley
- Slides adapted from CS60045 ARTIFICIAL INTELLIGENCE