CS 343H: Honors AI

Lecture 11: MDPs II

2/20/2014

Kristen Grauman

UT Austin

Slides courtesy of Dan Klein, UC Berkeley

Unless otherwise noted

Announcements

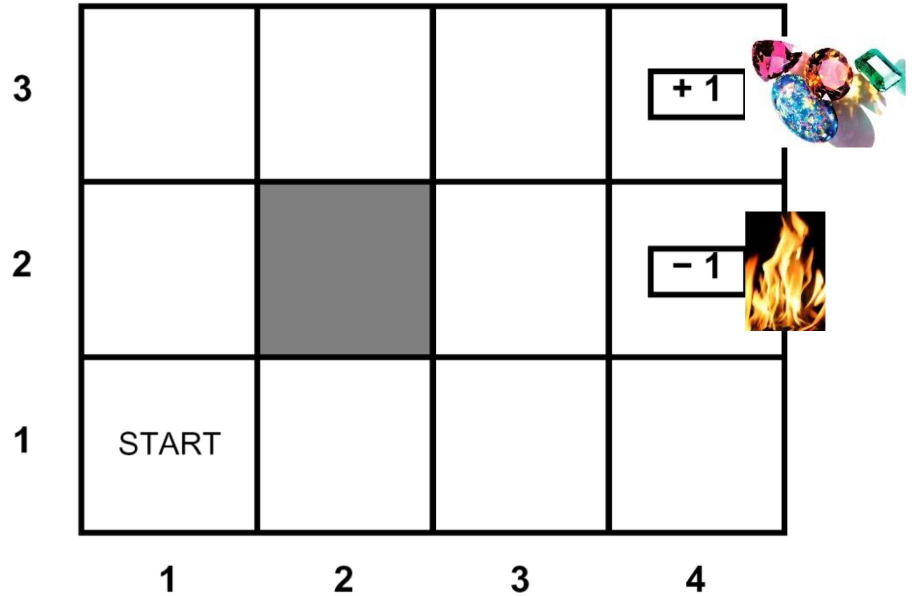
- Midterm is in class Thurs 3/6
  - Will provide practice problems next Thurs
- PS3 is posted, due on 3/20
  - OK to work in pairs for this one if you use "pair programming" practices
  - This explicitly means NOT dividing work between partners
  - Assignments help prep for exams
  - Submit just one code package, note the partners in the README

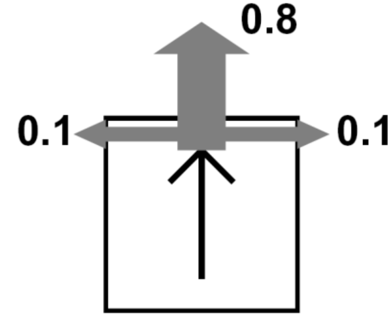
Today

- Recap of MDPs and value iteration
- Policy iteration
- Transition to reinforcement learning

Recall: Grid World

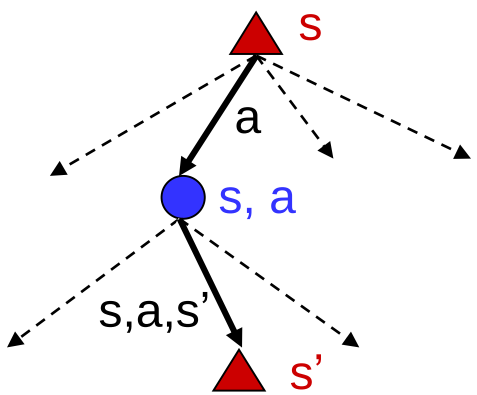
- The agent lives in a grid
- Walls block the agent's path
- The agent's actions do not always go as planned:
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent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





Recap: MDPs

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s,a,s')$
    - Prob that a from s leads to s'
    - i.e., $P(s' \mid s,a)$
  - A reward function $R(s, a, s')$ and discount γ
  - A start state



- 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

- Define the value (utility) of a state s:

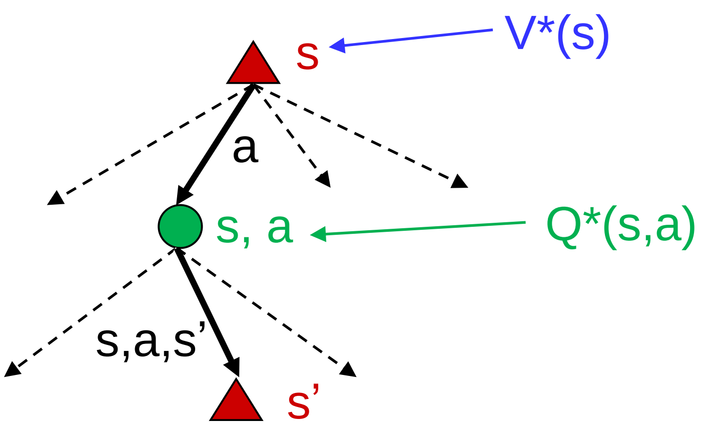
  $V^*(s)$ = expected utility starting in s and acting optimally

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

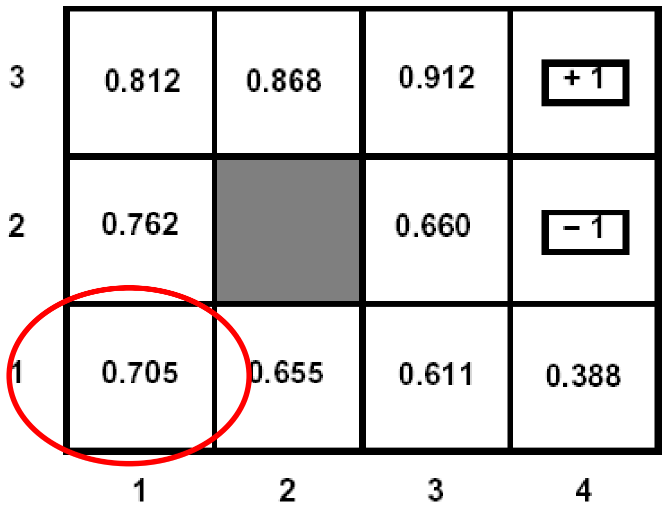
  $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally

- Define the optimal policy:

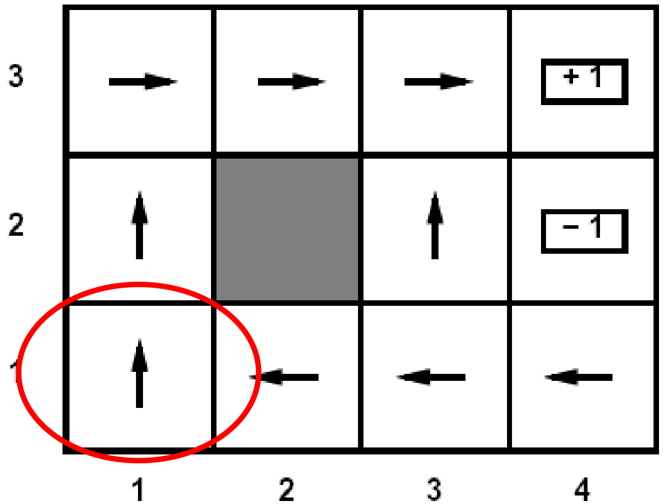
  $\pi^*(s)$ = optimal action from state s



Recall: Gridworld example



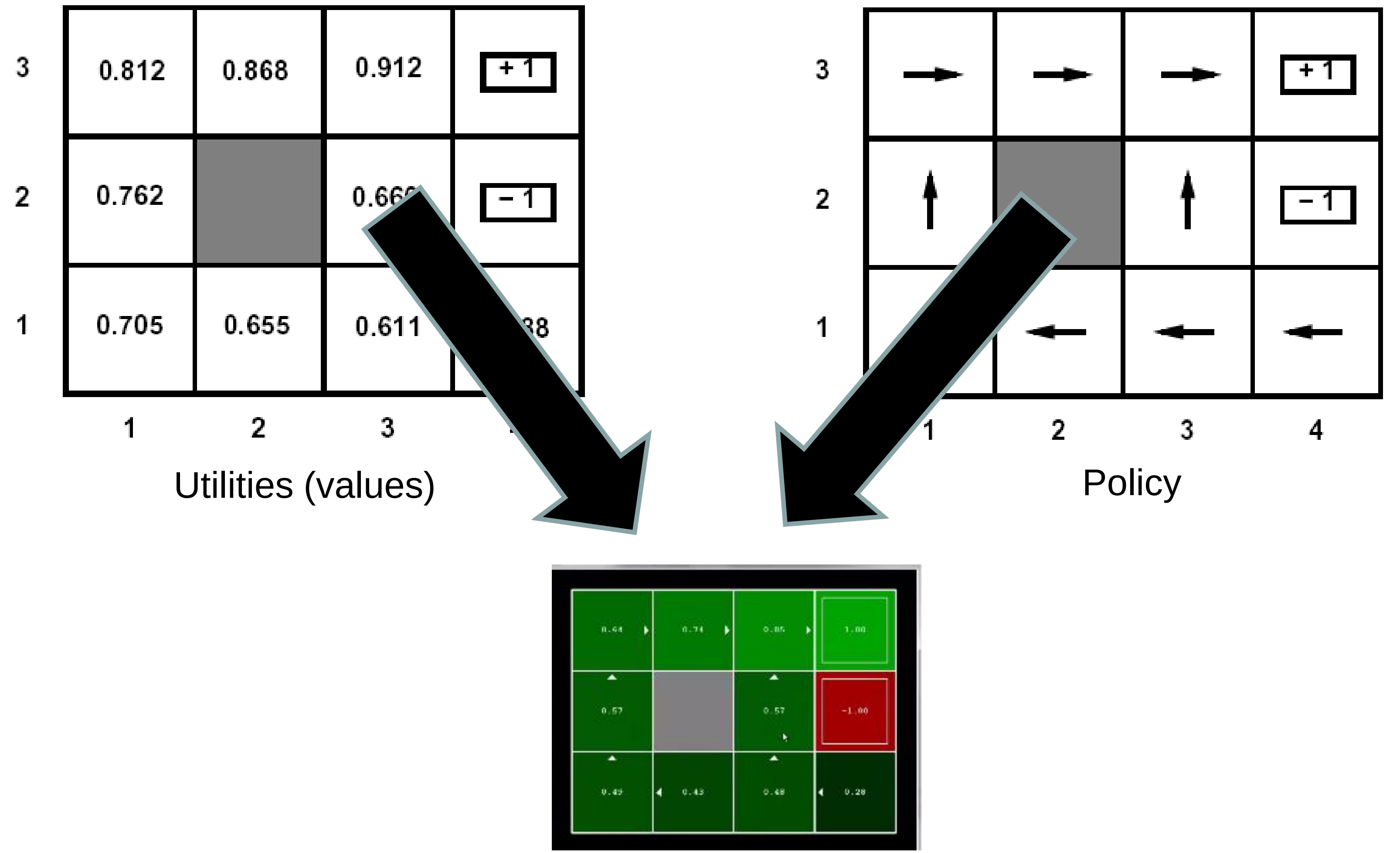
Utilities (values)



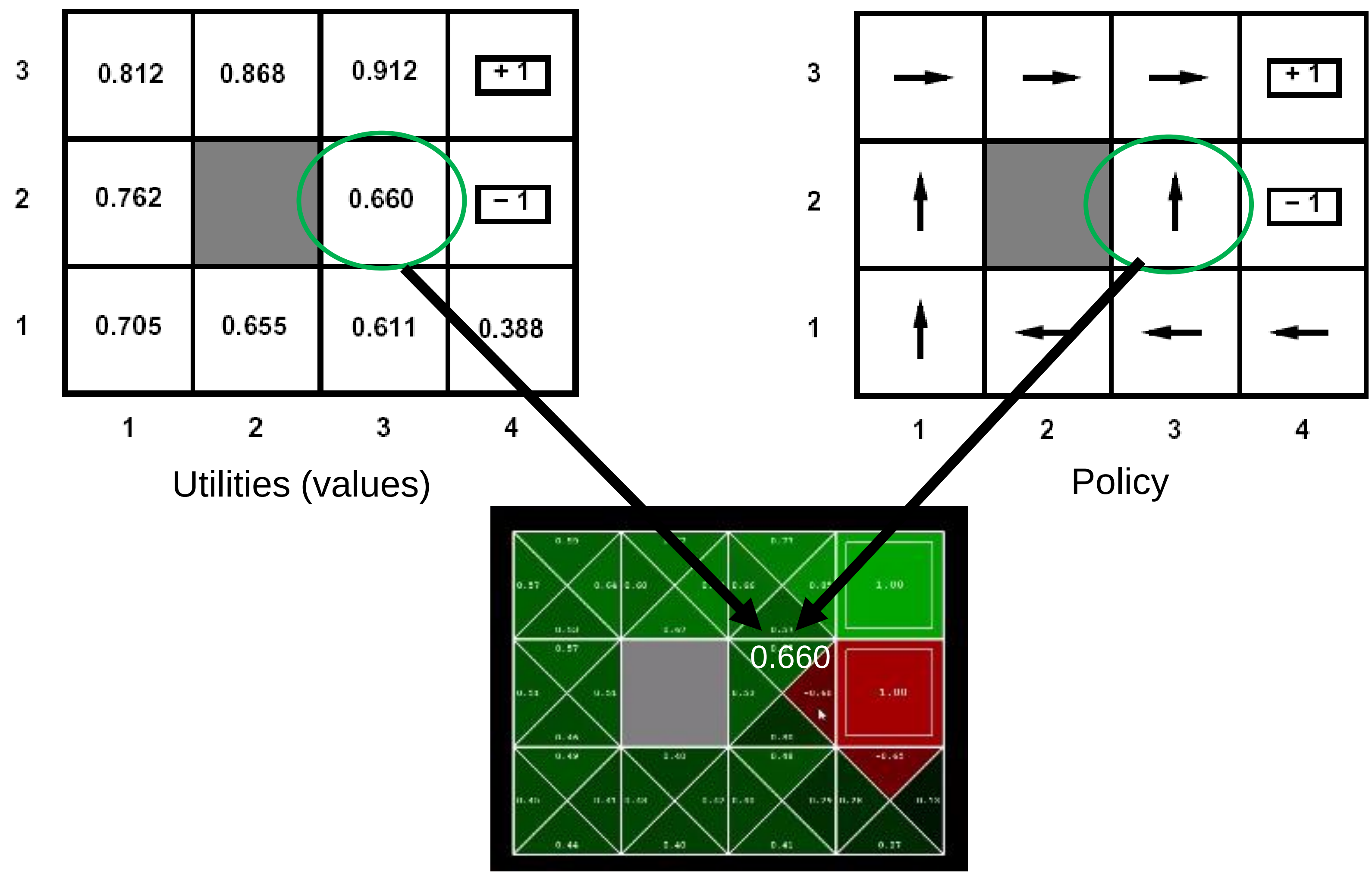
Policy

On average, we expect to get 0.705 if we went north in this state

Recall: Gridworld example



Recall: Gridworld example

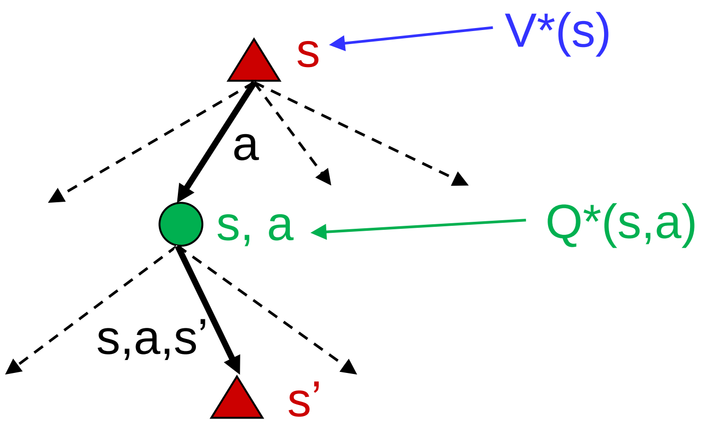


Q-values

Optimal quantities

- Define the value (utility) of a state s:
  $V^*(s)$ = expected utility starting in s and acting optimally
- Define 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
- Define the optimal policy:
  $\pi^*(s)$ = optimal action from state s

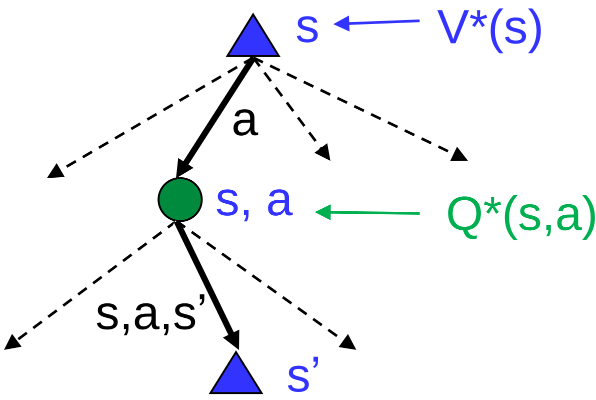
Cumulative – from that point forward



The Bellman Equations

- Definition of "optimal utility" leads to a simple one-step lookahead relationship amongst optimal utility values:

  Optimal rewards = maximize over first action and then follow optimal policy



- Formally:

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

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

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

Value Iteration

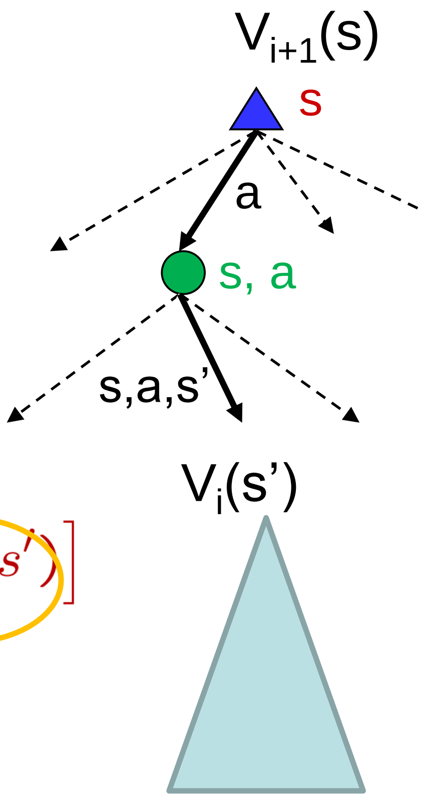
- Bellman equations characterize the optimal values:

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

- Value iteration computes them:

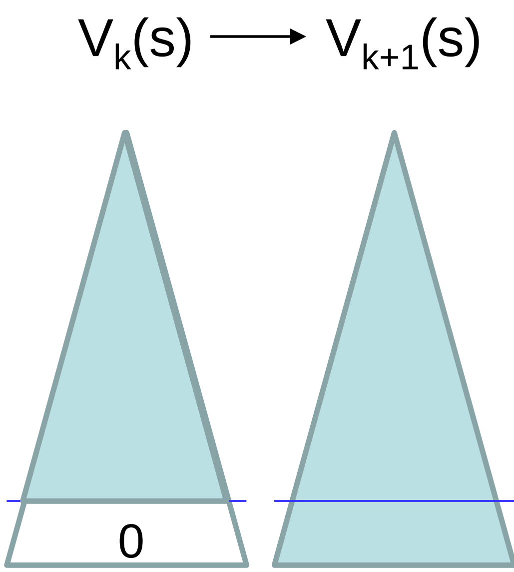
$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i(s')\right]$$

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



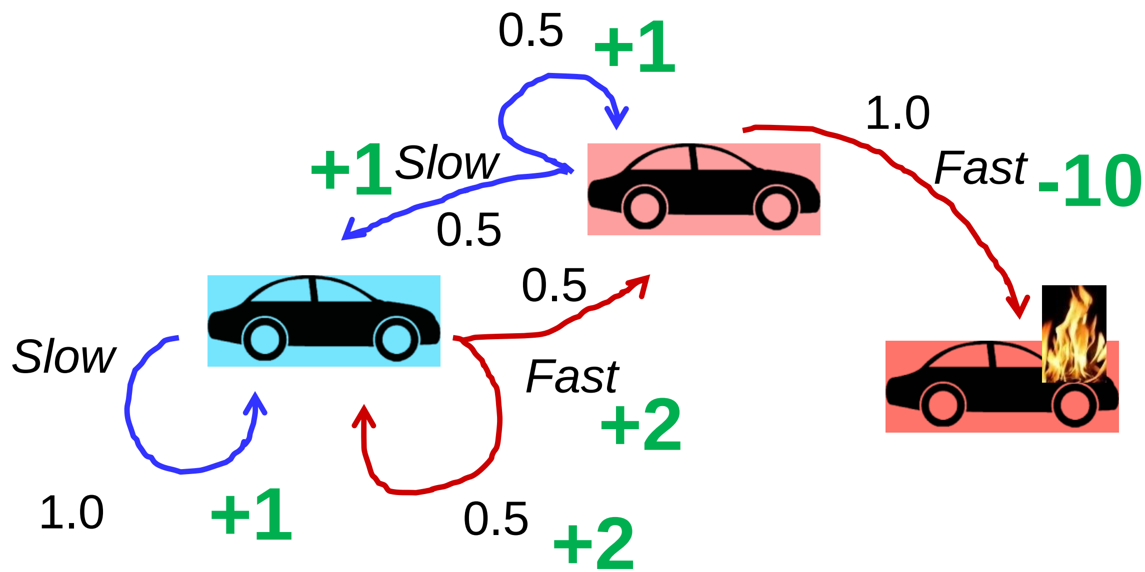
Convergence

- 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 resulting in nearly identical search trees.
  - The difference is that on the bottom layer, V_{k+1} has optimal 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 increase, the values converge



Example: value iteration

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



Assume no discount

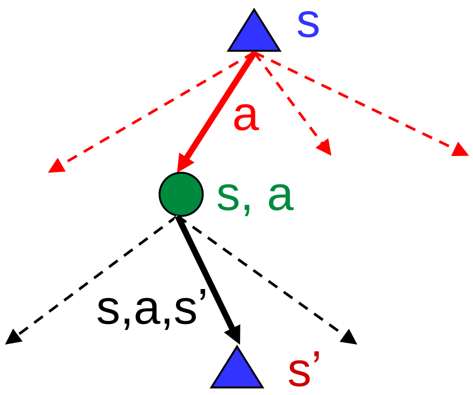
$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i(s') \right]$$

Today

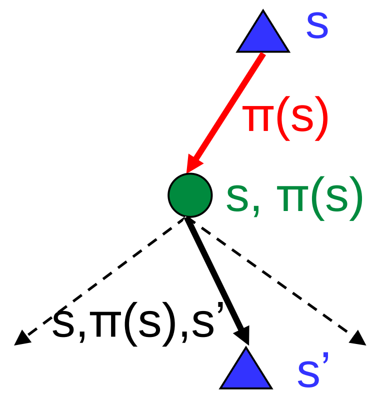
- Recap of MDPs and value iteration
- Policy iteration
- Transition to reinforcement learning

Fixed policies

Do the optimal action



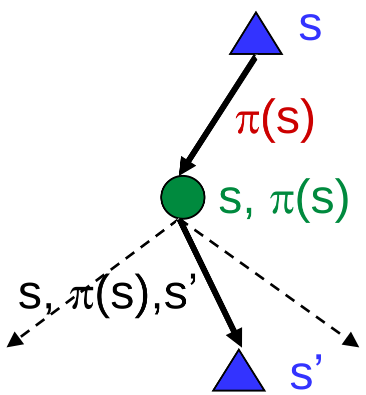
Do what π says to do



- 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 <u>fixed</u> (generally non-optimal) policy



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

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

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

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

Example: Policy evaluation

Always go right

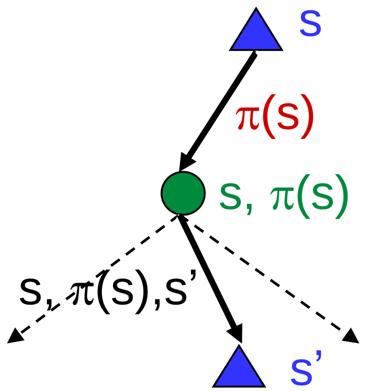
Example: Policy evaluation

Always go right

Always go forward

Policy Evaluation

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



- Idea 1: Turn recursive equations into updates

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

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

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

- Idea 2: Without the maxes, it's now a linear system
  - Solve with Matlab (or your favorite linear system solver)

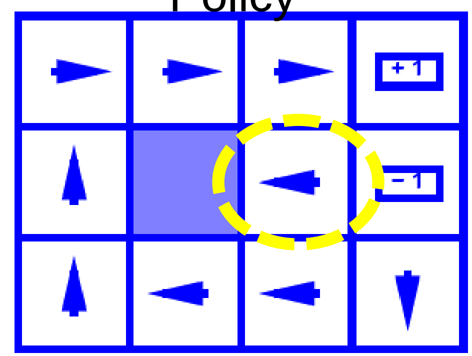
Policy extraction

Computing actions from values

Optimal values V*(s)

0.95	0.96	0.98	1.0
0.94		0.89	-1.0
0.92	0.91	0.90	0.80

Policy



How should we act? It's not obvious!

Computing actions from values

- Which action should we chose from state s:
  - Given optimal values V*?

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

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

Computing actions from values

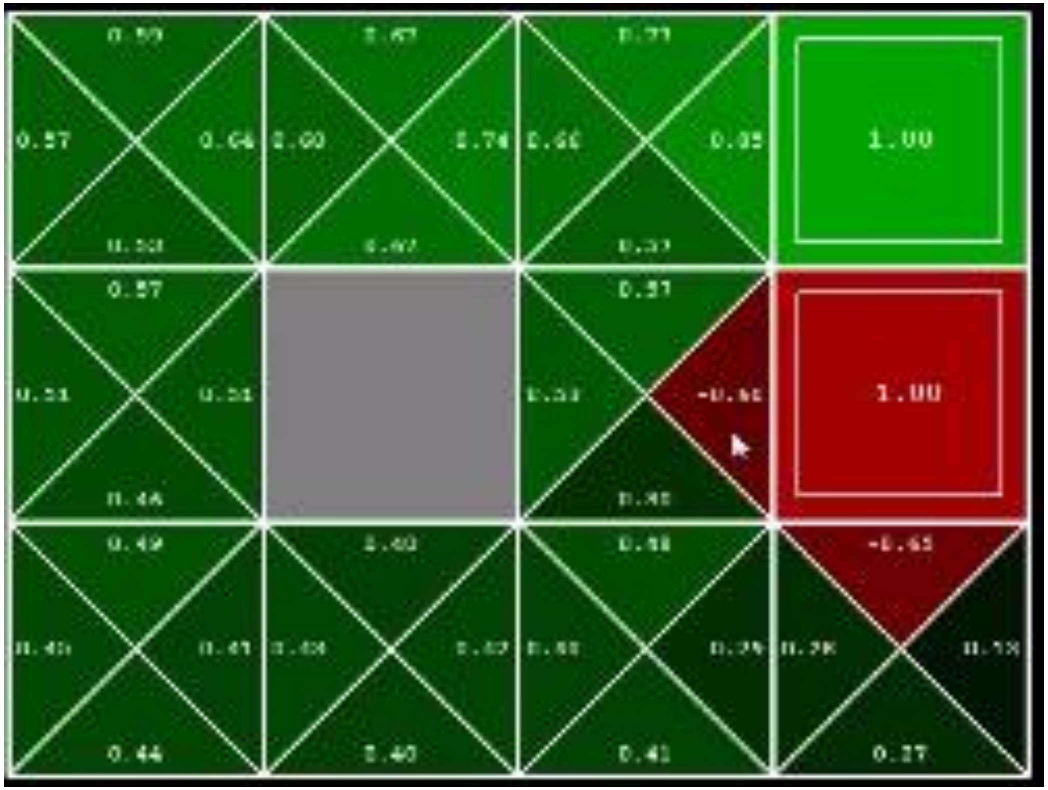
- Which action should we chose from state s:
  - Given optimal values V*?

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

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

Computing actions from Q-values

Optimal values Q*(a,s)



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

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

Computing actions from values

- Which action should we chose from state s:
  - Given optimal values V*?

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

  - Given optimal q-values Q?

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

  - Lesson: actions are easier to select from Q's!

Policy iteration

- Alternative approach for optimal values

Problems with Value Iteration

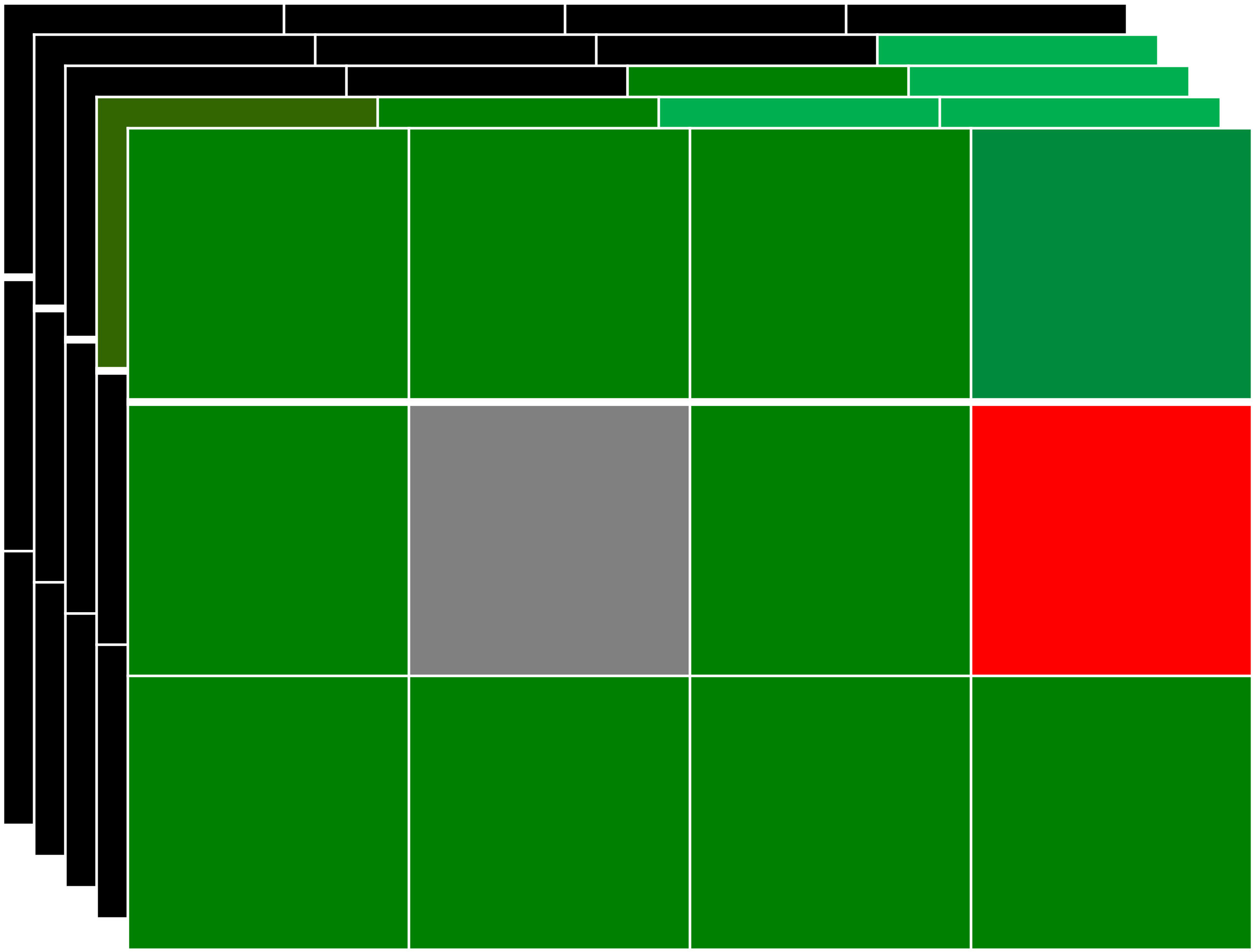
- Value iteration repeats the Bellman updates

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

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

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

Recall: Gridworld value iteration



As we increase k

Problems with Value Iteration

- Value iteration repeats the Bellman updates

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

- 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 do.

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 faster under some conditions

Policy Iteration

- Policy evaluation: with fixed current policy π, find values with simplified Bellman updates:
  - Iterate until values converge

$$V^{\pi_k}_{i+1}(s) \leftarrow \sum_{s'} T(s, \pi_k(s), s') \left[R(s, \pi_k(s), s') + \gamma V^{\pi_k}_i(s')\right]$$

- Policy improvement: with fixed utilities, find the best action according to one-step look-ahead

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

Comparison: value vs. policy iteration

- Both compute the same thing (optimal values for all states)
- In value iteration:
  - Every iteration updates both the values (explicitly, based on current utilities) and policy (implicitly, based on current utilities)
  - Tracking the policy isn't necessary; taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes to update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After 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

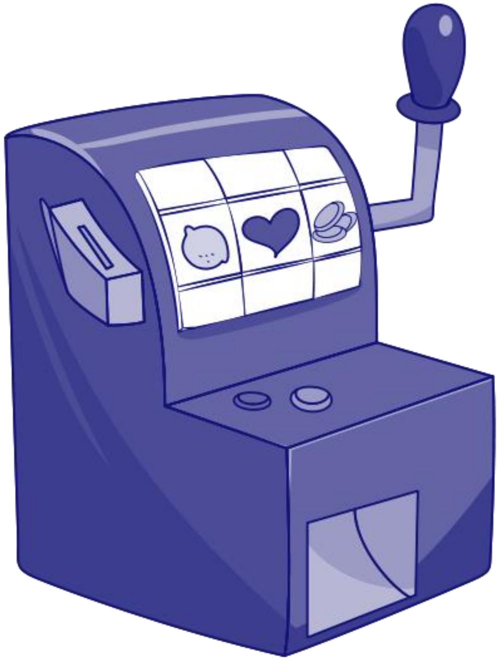
- So you want to…
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

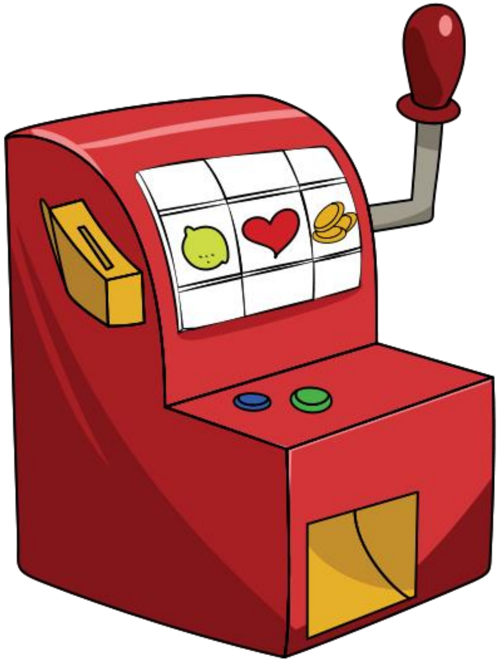
- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions

Today

- Recap of MDPs and value iteration
- Policy iteration
- Transition to reinforcement learning

Double bandits





- Which slot machine should we play?

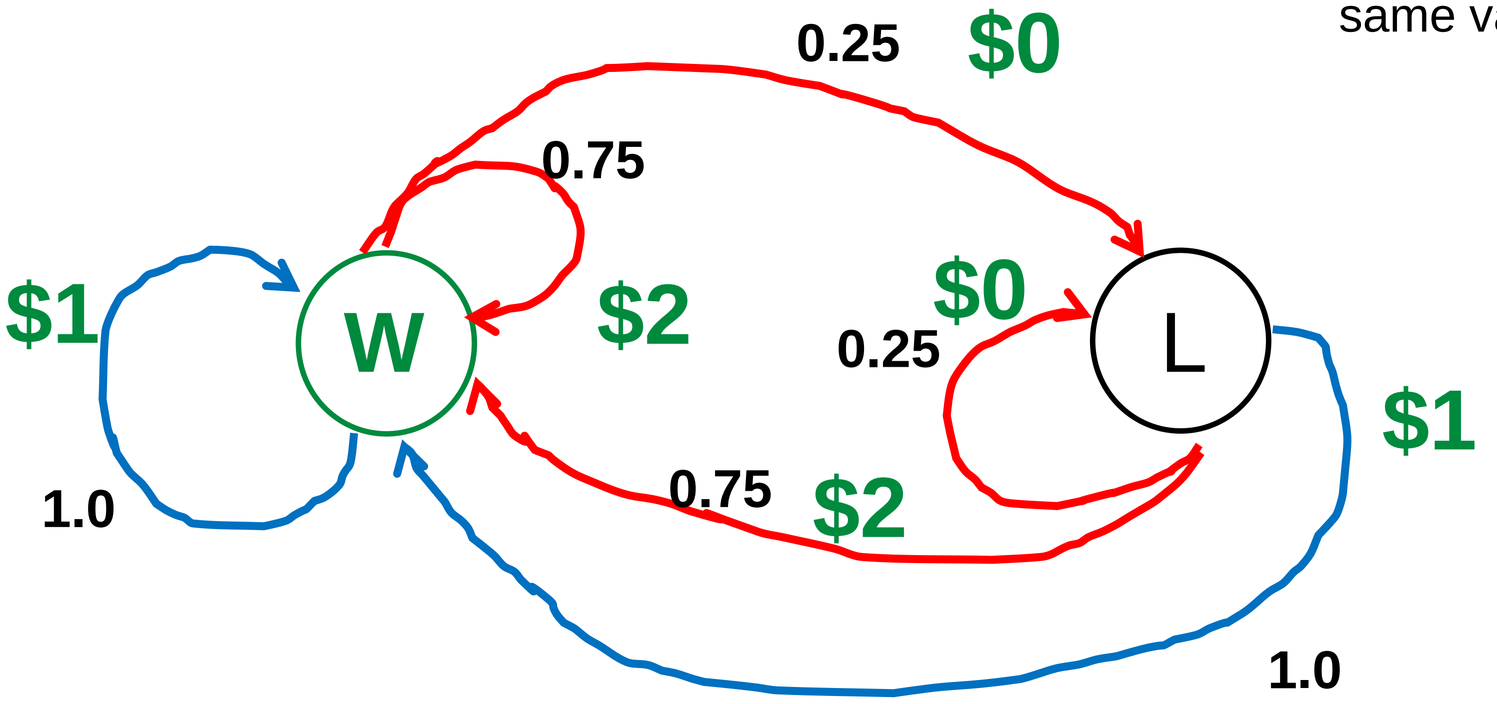
Double-bandit MDP

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

No discount

100 time steps

Both states have the same value



Offline planning

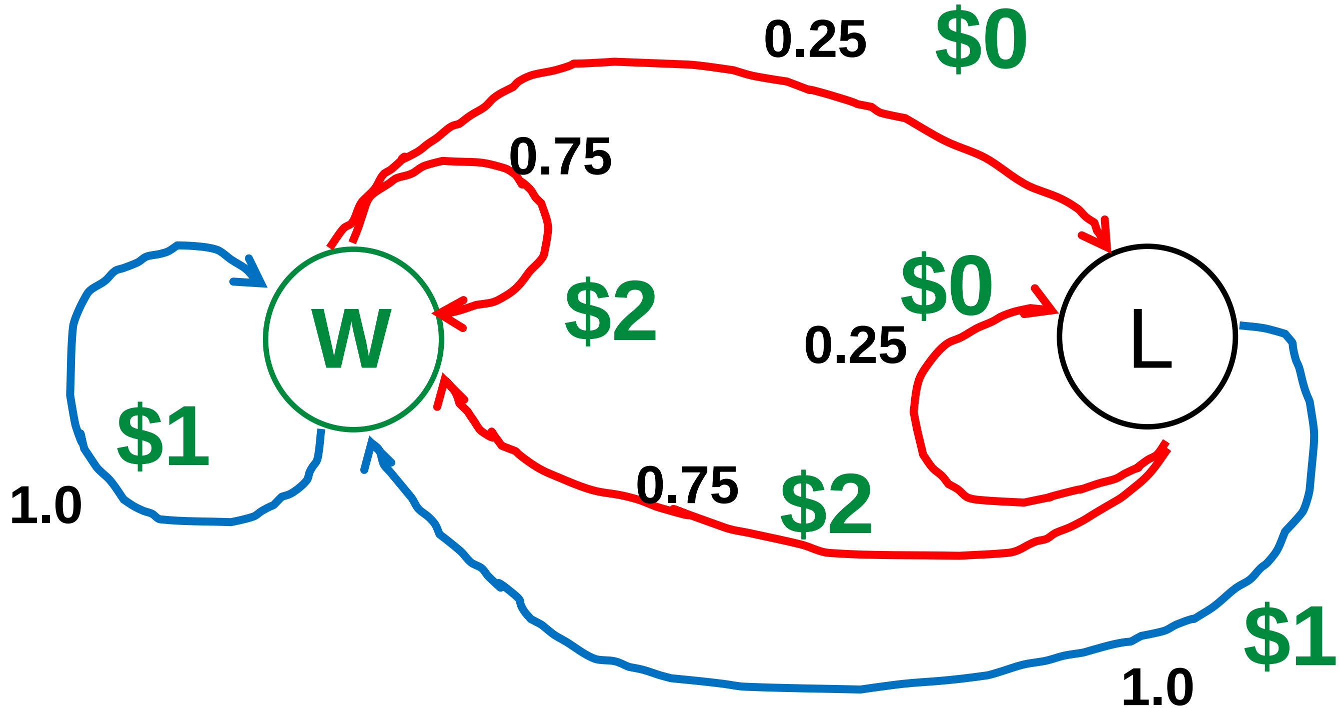
- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

No discount

100 time steps

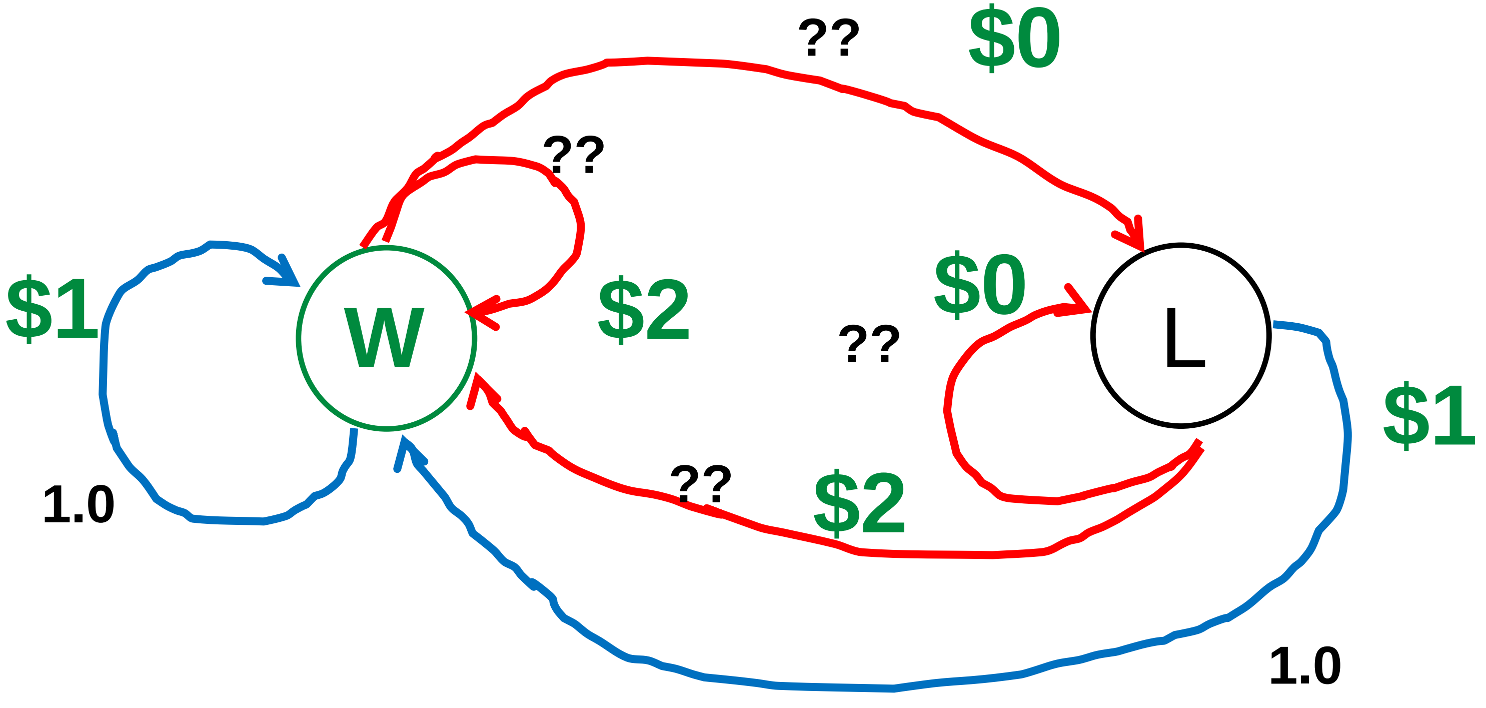
Both states have the same value

	Value
Play Blue	100
Play Red	150



Online planning

- Rules changed! Red's win chance is different



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