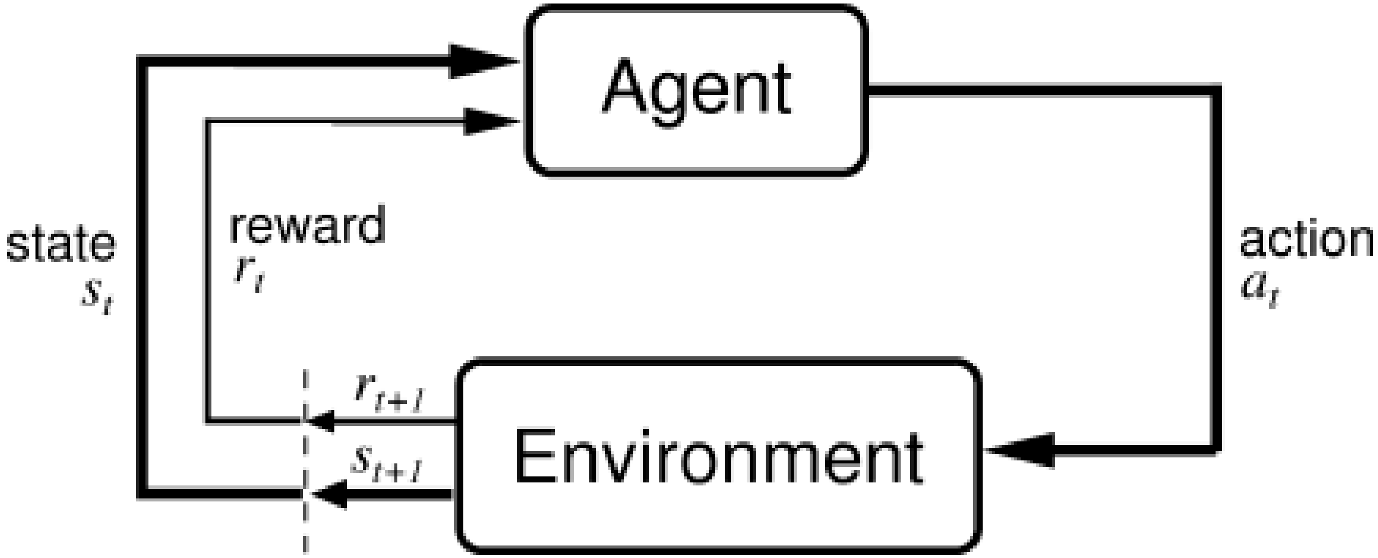
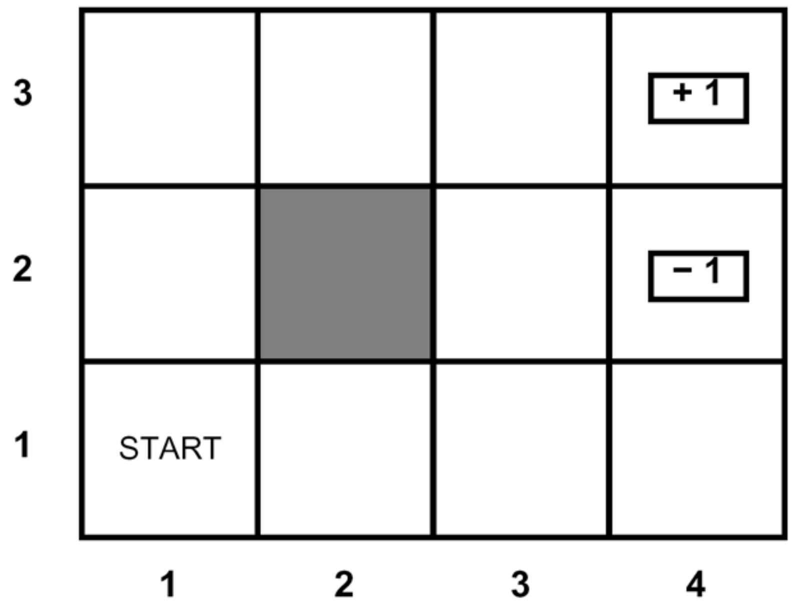
Reinforcement Learning

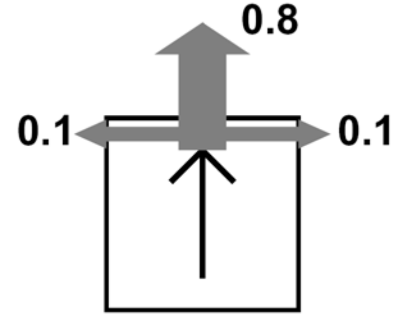
- Basic idea:
  - Receive feedback in the form of rewards
  - Agent's utility is defined by the reward function
  - Must (learn to) act so as to maximize expected rewards



Grid World

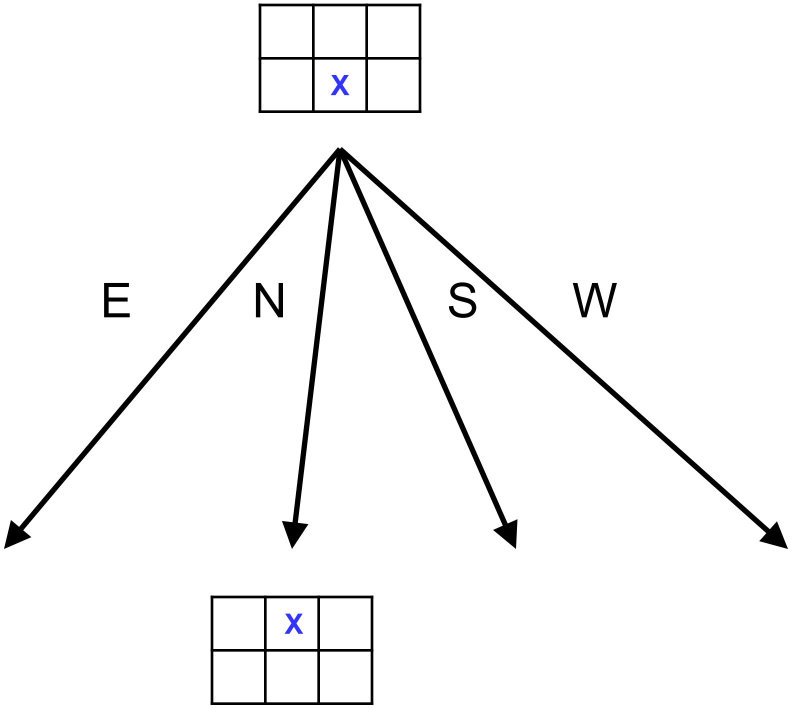
- The agent lives in a grid
- Walls block the agent's path
- The agent's actions do not always go as planned:
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent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



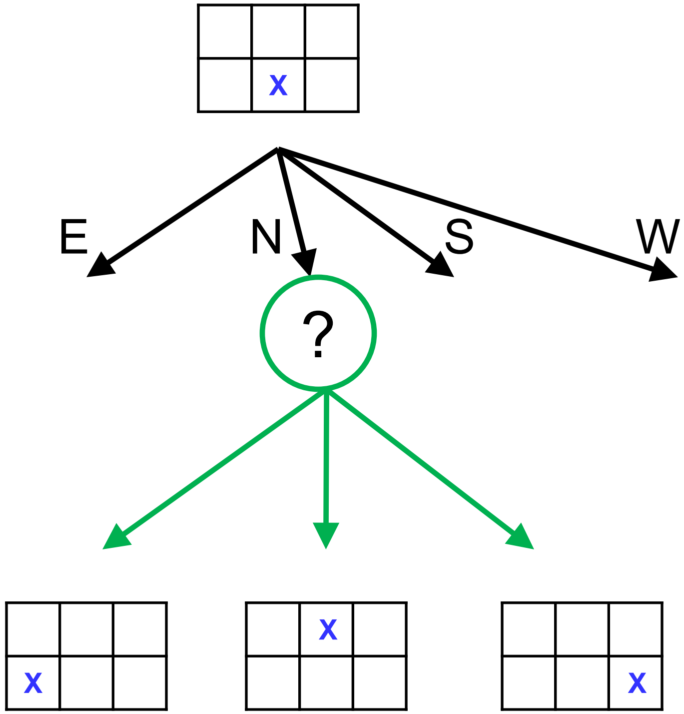


Grid Futures

Deterministic Grid World

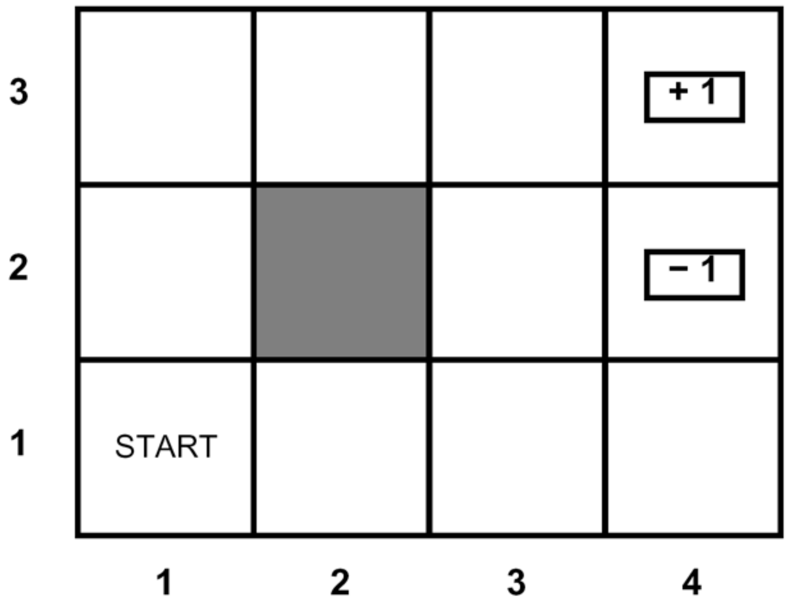


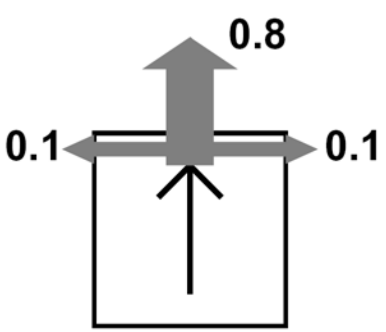
Stochastic Grid World



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')
    - Prob that a from s leads to s'
    - i.e., P(s' | s,a)
    - Also called the model
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
  - A start state (or distribution)
  - Maybe a terminal state
- MDPs are a family of non-deterministic search problems
  - Reinforcement learning: MDPs where we don't know the transition or reward functions





Keepaway

- http://www.cs.utexas.edu/~AustinVilla/sim/keepaway/swf/learn360.swf

- SATR
- S_0, S_0

What is Markov about MDPs?

- Andrey Markov (1856-1922)
- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means:

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

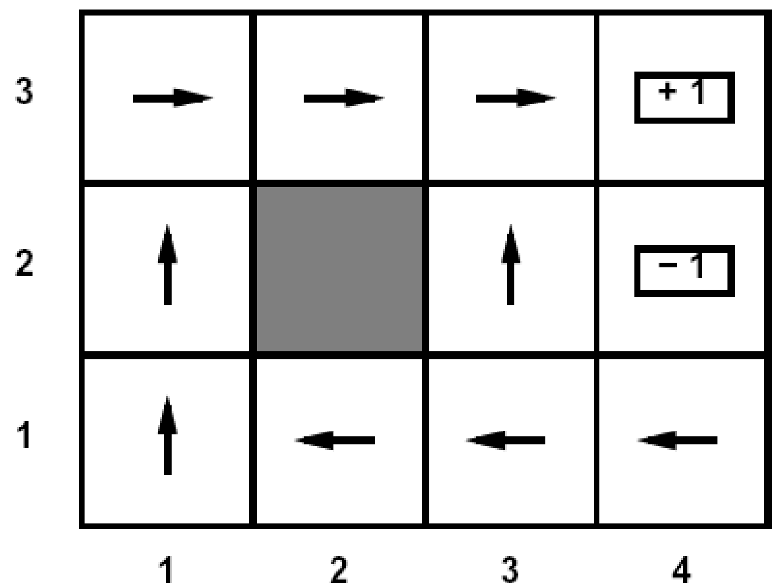
$$=$$

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

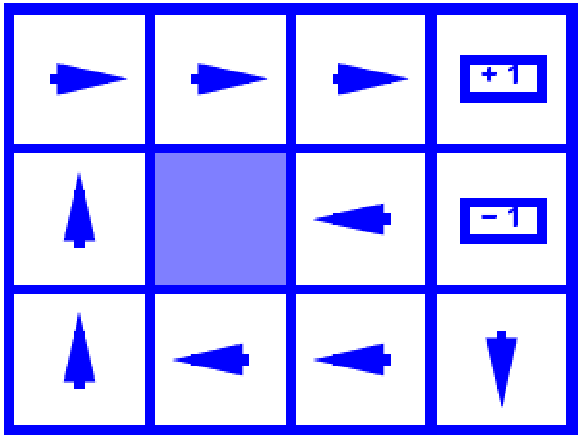
Solving MDPs

- In deterministic single-agent search problems, want an optimal plan, or sequence of actions, from start to a goal
- In an MDP, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy maximizes expected utility if followed
  - Defines a reflex agent

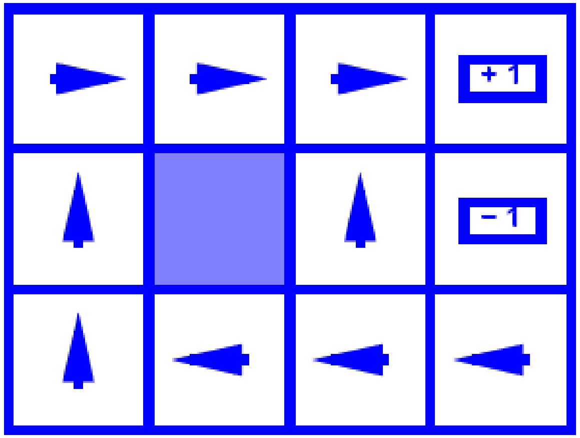
Optimal policy when R(s, a, s') = -0.03 for all non-terminals s



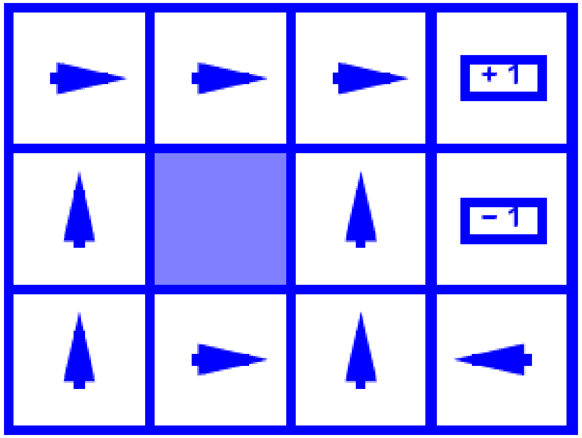
Example Optimal Policies



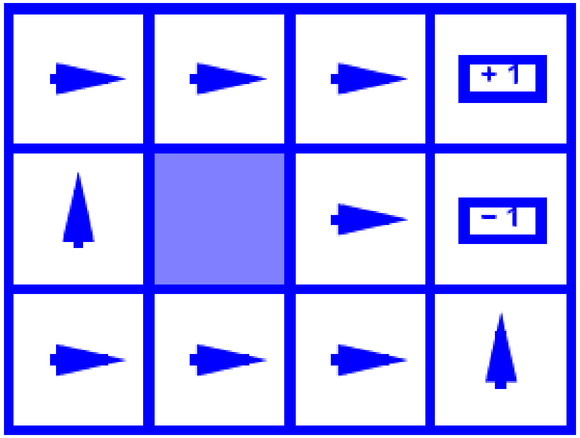
R(s) = -0.01



R(s) = -0.03



R(s) = -0.4



R(s) = -2.0

Utilities of Sequences

- In order to formalize optimality of a policy, need to understand utilities of sequences of rewards
- Typically consider stationary preferences:

$$[r, r_0, r_1, r_2, \ldots] \succ [r, r'_0, r'_1, r'_2, \ldots]$$
$$\Leftrightarrow$$
$$[r_0, r_1, r_2, \ldots] \succ [r'_0, r'_1, r'_2, \ldots]$$

- Theorem: only two ways to define stationary utilities
  - Additive utility:

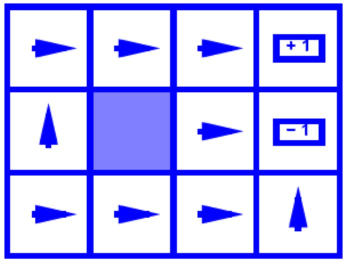
$$U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$$

  - Discounted utility:

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

Infinite Utilities?!

- Problem: infinite state sequences have infinite rewards



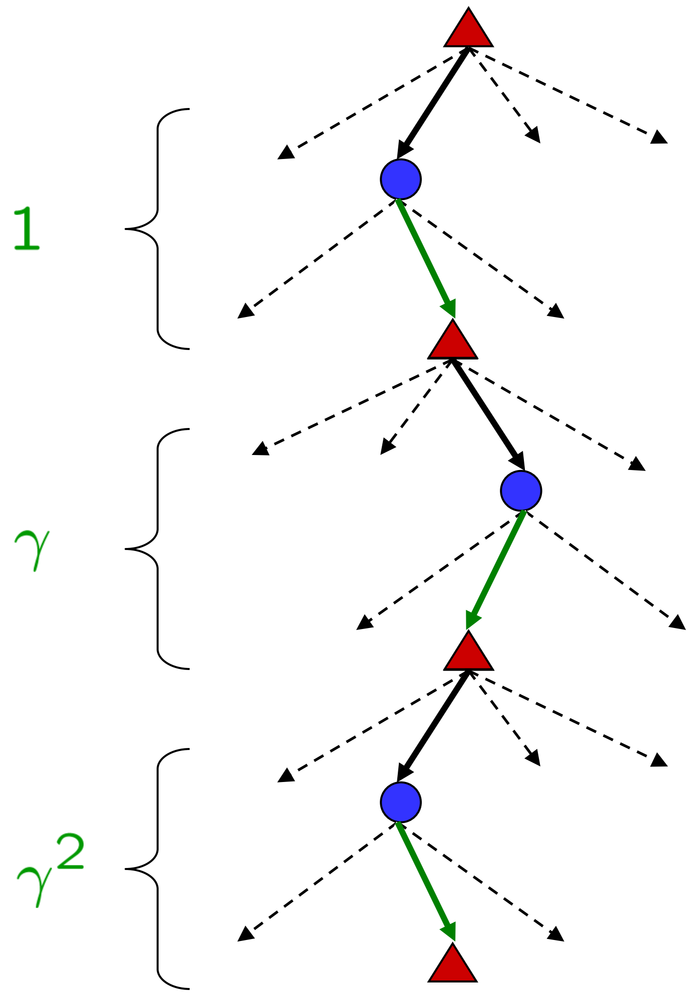
- Solutions:
  - Finite horizon:
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "done" for High-Low)
  - Discounting: for $0 < \gamma < 1$

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

    - Smaller γ means smaller "horizon" – shorter term focus

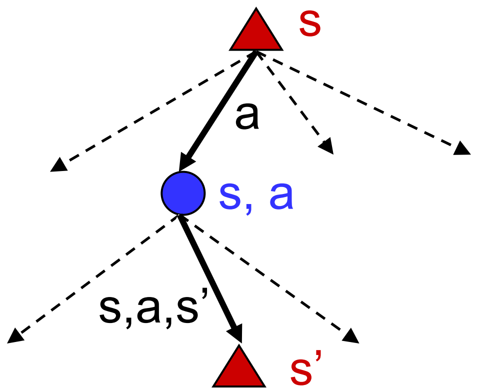
Discounting

- Typically discount rewards by $\gamma < 1$ each time step
  - Sooner rewards have higher utility than later rewards
  - Also helps the algorithms converge



Recap: Defining MDPs

- Markov decision processes:
  - States S
  - Start state s_0
  - 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 (or return) = sum of discounted rewards

Optimal Utilities

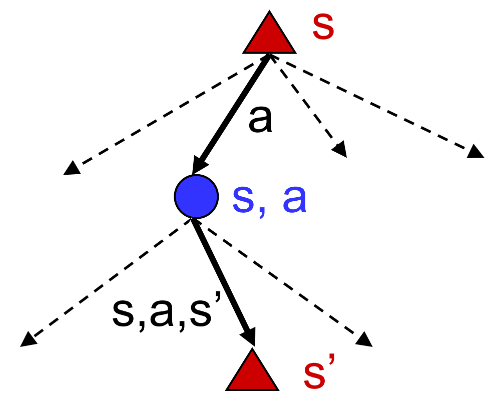
- Fundamental operation: compute the values (optimal expectimax utilities) of states s

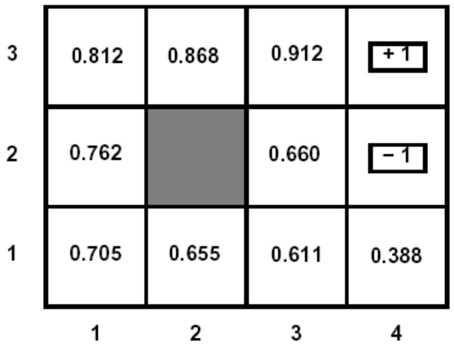
- Why? Optimal values define optimal policies!

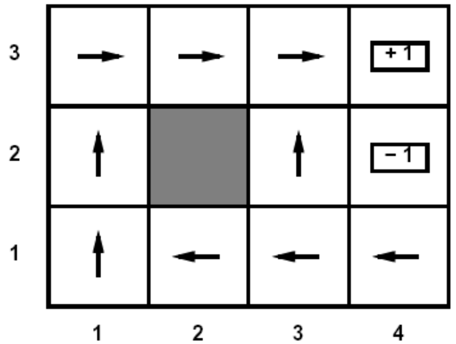
- Define the value of a state s:
  $V^*(s)$ = expected utility starting in s and acting optimally

- Define the value of a q-state (s,a):
  $Q^*(s,a)$ = expected utility starting in s, taking action a and thereafter acting optimally

- Define the optimal policy:
  $\pi^*(s)$ = optimal action from state s



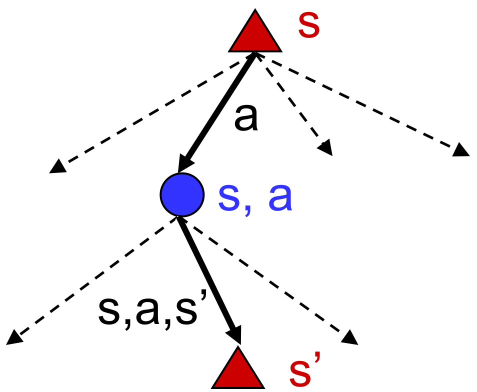




The Bellman Equations

- 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:

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

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

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

Solving MDPs

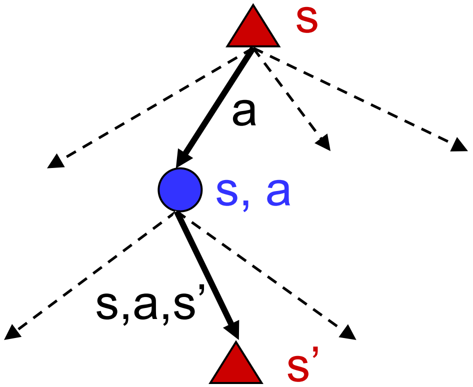
- We want to find the optimal policy π^*

- Proposal 1: modified expectimax search, starting from each state s:

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

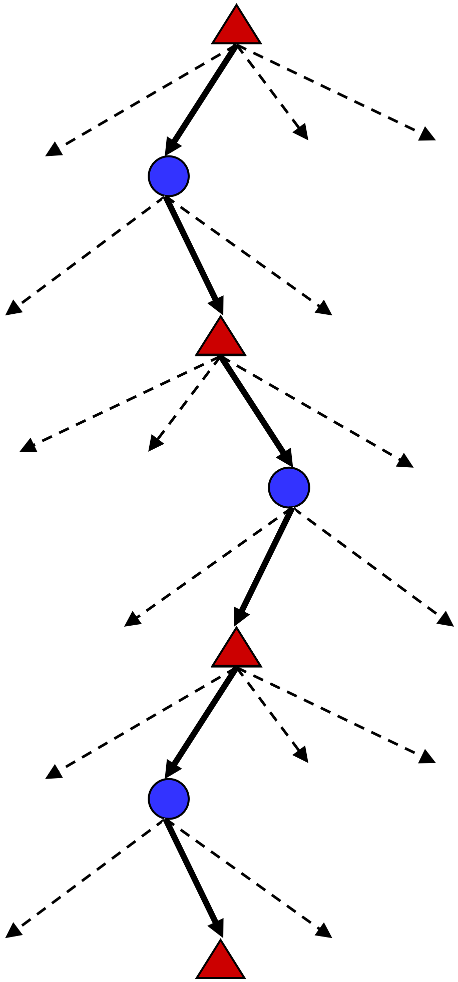
$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

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



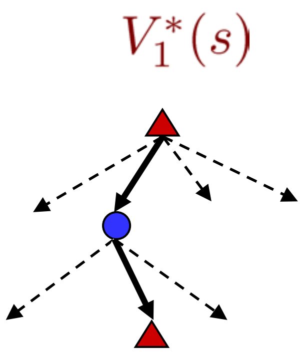
Why Not Search Trees?

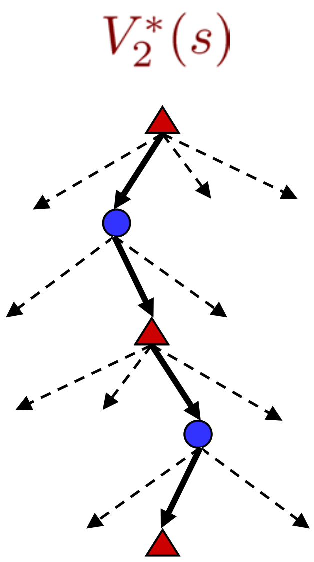
- Why not solve with expectimax?
- Problems:
  - This tree is usually infinite (why?)
  - Same states appear over and over (why?)
  - We would search once per state (why?)
- Idea: Value iteration
  - Compute optimal values for all states all at once using successive approximations
  - Will be a bottom-up dynamic program similar in cost to memoization
  - Do all planning offline, no replanning needed!



Value Estimates

- Calculate estimates $V_k^*(s)$
  - Not the optimal value of s!
  - The optimal value considering only next k time steps (k rewards)
  - As $k \to \infty$, it approaches the optimal value
- Almost solution: recursion (i.e. expectimax)
- Correct solution: dynamic programming





Value Iteration

- Idea:
  - Start with $V_0^*(s) = 0$, which we know is right (why?)
  - Given V_i^*, calculate the values for all states for depth i+1:

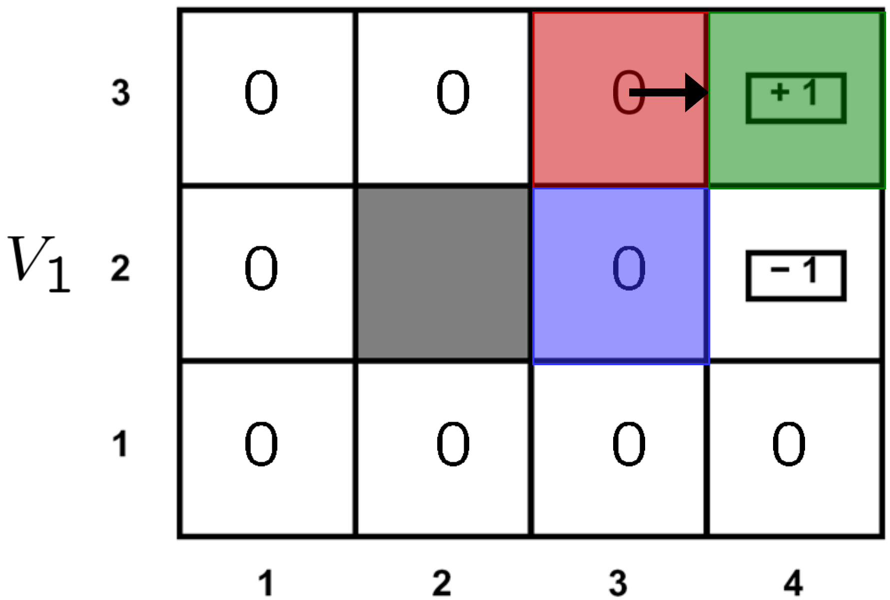
$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i(s') \right]$$

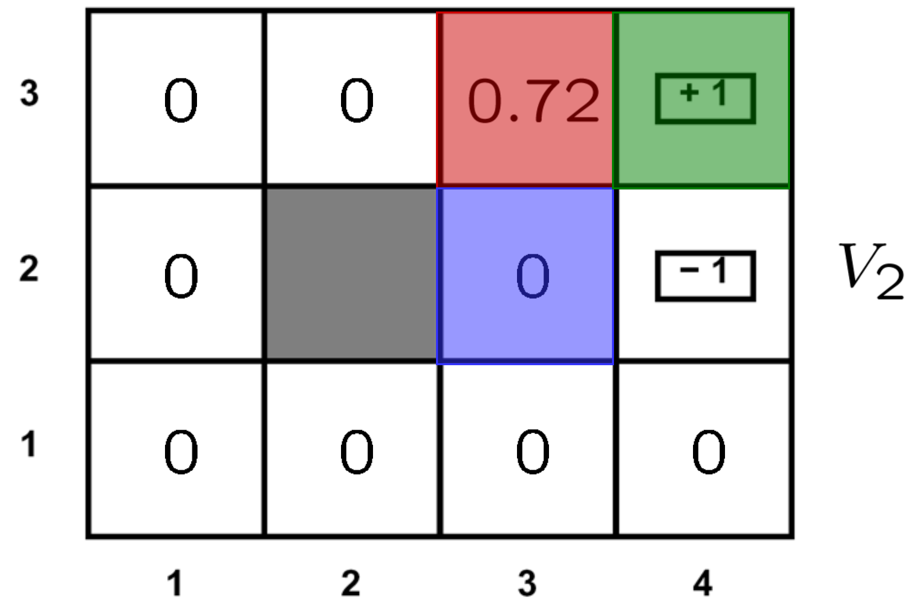
  - This is called a value update or Bellman update
  - Repeat until convergence

- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

Example: γ=0.9, living reward=0, noise=0.2

Example: Bellman Updates





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

$$V_2(\langle 3,3 \rangle) = \sum_{s'} T(\langle 3,3 \rangle, \text{right}, s') \left[R(\langle 3,3 \rangle) + 0.9\, V_1(s') \right]$$

$$= 0.9\,[0.8 \cdot 1 + 0.1 \cdot 0 + 0.1 \cdot 0]$$

max happens for a=right, other actions not shown

Example: Value Iteration

V_2

3	0	0	0.72	+1
2	0		0	−1
1	0	0	0	0
	1	2	3	4

V_3

3	0	0.52	0.78	+1
2	0		0.43	−1
1	0	0	0	0
	1	2	3	4

- Information propagates outward from terminal states and eventually all states have correct value estimates

Convergence*

- Define the max-norm: $||U|| = \max_s |U(s)|$

- Theorem: For any two approximations U and V

$$||U^{t+1} - V^{t+1}|| \leq \gamma \, ||U^t - V^t||$$

  - I.e. any distinct approximations must get closer to each other, so, in particular, any approximation must get closer to the true U and value iteration converges to a unique, stable, optimal solution

- Theorem:

$$||U^{t+1} - U^t|| < \epsilon, \Rightarrow ||U^{t+1} - U|| < 2\epsilon\gamma/(1-\gamma)$$

  - I.e. once the change in our approximation is small, it must also be close to correct

Practice: Computing Actions

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

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

  - Given optimal q-values Q?

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

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

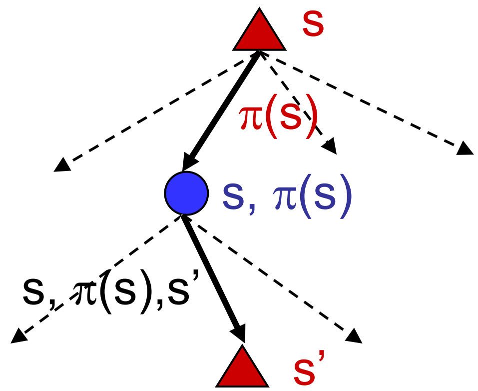
Utilities for Fixed Policies

- Another basic operation: compute the utility of a state s under a fix (general 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):



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

Value Iteration

- Idea:
  - Start with $V_0^*(s) = 0$, which we know is right (why?)
  - Given V_i^*, calculate the values for all states for depth i+1:

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

  - This is called a value update or Bellman update
  - Repeat until convergence

- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

Policy Iteration

- Problem with value iteration:
  - Considering all actions each iteration is slow: takes |A| times longer than policy evaluation
  - But policy doesn't change each iteration, time wasted

- Alternative to value iteration:
  - Step 1: Policy evaluation: calculate utilities for a fixed policy (not optimal utilities!) until convergence (fast)
  - Step 2: Policy improvement: update policy using one-step lookahead with resulting converged (but not optimal!) utilities (slow but infrequent)
  - 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

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

- 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

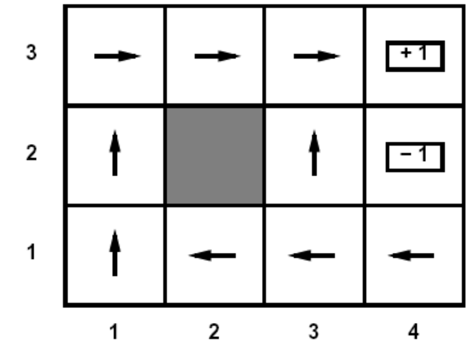
- In value iteration:
  - Every pass (or “backup”) updates both utilities (explicitly, based on current utilities) and policy (possibly implicitly, based on current policy)
- In policy iteration:
  - Several passes to update utilities with frozen policy
  - Occasional passes to update policies
- Hybrid approaches (asynchronous policy iteration):
  - Any sequences of partial updates to either policy entries or utilities will converge if every state is visited infinitely often

Reinforcement Learning

- Reinforcement learning:
  - Still assume an MDP:
    - A set of states $s \in S$
    - A set of actions (per state) A
    - A model T(s,a,s')
    - A reward function R(s,a,s')
  - Still looking for a policy $\pi(s)$
  - New twist: don't know T or R
    - i.e. don't know which states are good or what the actions do
    - Must actually try actions and states out to learn

Demo: Robot Dogs!

Passive Learning



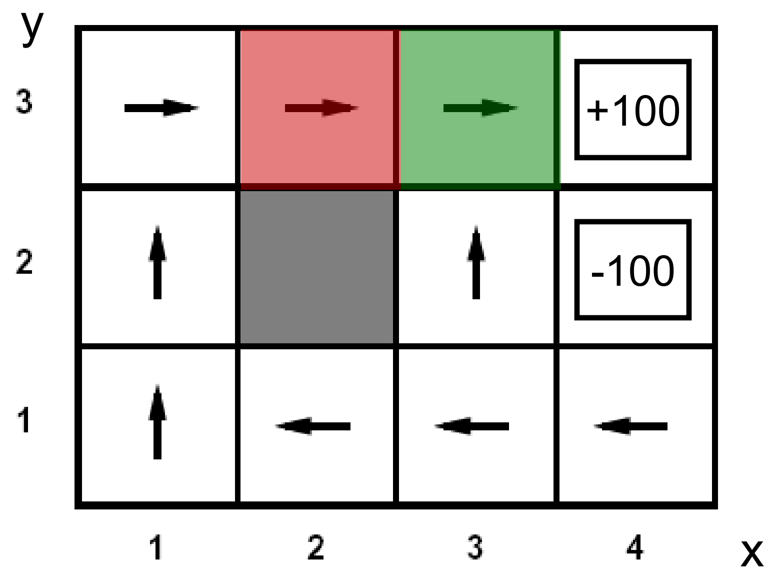
- Simplified task
  - You don’t know the transitions T(s,a,s’)
  - You don’t know the rewards R(s,a,s’)
  - You are given a policy π(s)
  - Goal: learn the state values
  - what policy evaluation did
- In this case:
  - Learner “along for the ride”
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - We’ll get to the active case soon
  - This is NOT offline planning! You actually take actions in the world and see what happens

Example: Direct Evaluation

- Episodes:

(1,1) up -1
(1,2) up -1
(1,2) up -1
(1,3) right -1
(2,3) right -1
(3,3) right -1
(3,2) up -1
(3,3) right -1
(4,3) exit +100
(done)

(1,1) up -1
(1,2) up -1
(1,3) right -1
(2,3) right -1
(3,3) right -1
(3,2) up -1
(4,2) exit -100
(done)



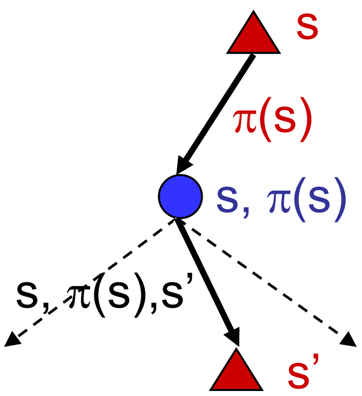
γ = 1, R = -1

V(2,3) ~ (96 + -103) / 2 = -3.5

V(3,3) ~ (99 + 97 + -102) / 3 = 31.3

Recap: Model-Based Policy Evaluation

- Simplified Bellman updates to calculate V for a fixed policy:
  - New V is expected one-step-look-ahead using current V
  - Unfortunately, need T and R

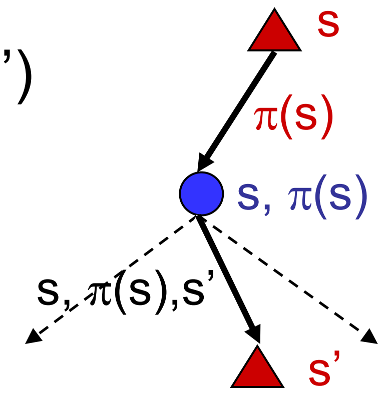


$$V_0^\pi(s) = 0$$

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

Model-Based Learning

- Idea:
  - Learn the model empirically through experience
  - Solve for values as if the learned model were correct
- Simple empirical model learning
  - Count outcomes for each s,a
  - Normalize to give estimate of **T(s,a,s')**
  - Discover **R(s,a,s')** when we experience (s,a,s')
- Solving the MDP with the learned model
  - Iterative policy evaluation, for example



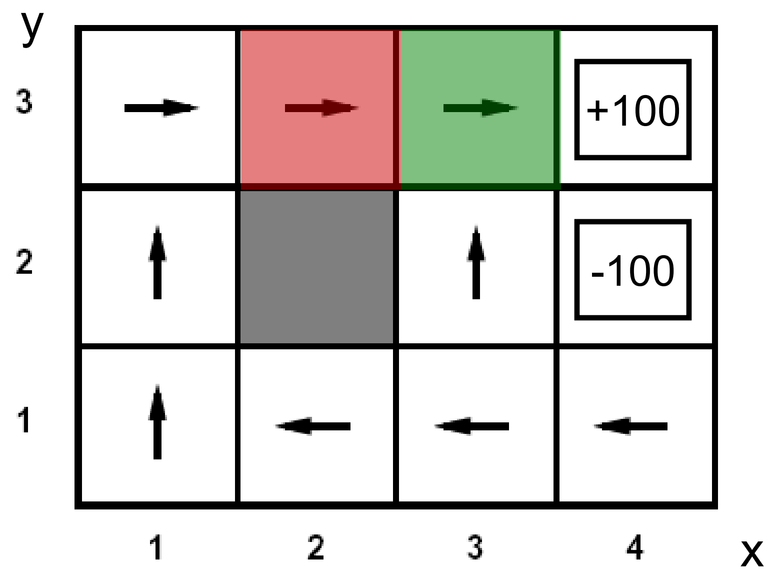
$$V_{i+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^{\pi}(s')]$$

Example: Model-Based Learning

- Episodes:

(1,1) up -1
(1,2) up -1
(1,2) up -1
(1,3) right -1
(2,3) right -1
(3,3) right -1
(3,2) up -1
(3,3) right -1
(4,3) exit +100
(done)

(1,1) up -1
(1,2) up -1
(1,3) right -1
(2,3) right -1
(3,3) right -1
(3,2) up -1
(4,2) exit -100
(done)



$\gamma = 1$

T(<3,3>, right, <4,3>) = 1 / 3

T(<2,3>, right, <3,3>) = 2 / 2

Model-Free Learning

- Want to compute an expectation weighted by P(x):

$$E[f(x)] = \sum_x P(x) f(x)$$

- Model-based: estimate P(x) from samples, compute expectation

$$x_i \sim P(x)$$

$$\hat{P}(x) = \text{count}(x)/k$$

$$E[f(x)] \approx \sum_x \hat{P}(x) f(x)$$

- Model-free: estimate expectation directly from samples

$$x_i \sim P(x)$$

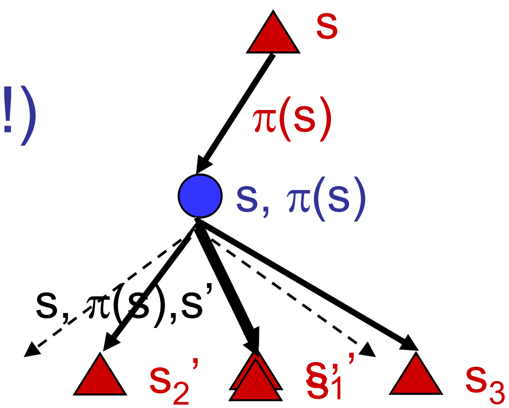
$$E[f(x)] \approx \frac{1}{k} \sum_i f(x_i)$$

- Why does this work? Because samples appear with the right frequencies!

Sample-Based Policy Evaluation?

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

- Who needs T and R? Approximate the expectation with samples (drawn from T!)



$$sample_1 = R(s, \pi(s), s_1') + \gamma V_i^{\pi}(s_1')$$

$$sample_2 = R(s, \pi(s), s_2') + \gamma V_i^{\pi}(s_2')$$

$$\ldots$$

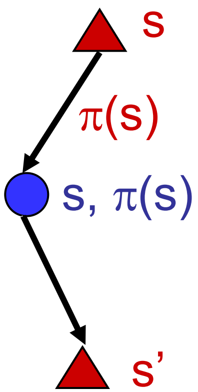
$$sample_k = R(s, \pi(s), s_k') + \gamma V_i^{\pi}(s_k')$$

$$V_{i+1}^{\pi}(s) \leftarrow \frac{1}{k} \sum_i sample_i$$

Almost! But we only actually make progress when we move to i+1.

Temporal-Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience (s,a,s',r)
  - Likely s' will contribute updates more often



- Temporal difference learning
  - Policy still fixed!
  - Move values toward value of whatever successor occurs: running average!

Sample of V(s): $$sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

Update to V(s): $$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$$

Same update: $$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

Exponential Moving Average

- Exponential moving average
  - Makes recent samples more important

$$\bar{x}_n = \frac{x_n + (1-\alpha)\cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + \dots}$$

  - Forgets about the past (distant past values were wrong anyway)
  - Easy to compute from the running average

$$\bar{x}_n = (1-\alpha)\cdot \bar{x}_{n-1} + \alpha \cdot x_n$$

- Decreasing learning rate can give converging averages

Example: TD Policy Evaluation

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

(1,1) up -1

(1,2) up -1

(1,2) up -1

(1,3) right -1

(2,3) right -1

(3,3) right -1

(3,2) up -1

(3,3) right -1

(4,3) exit +100

(done)

(1,1) up -1

(1,2) up -1

(1,3) right -1

(2,3) right -1

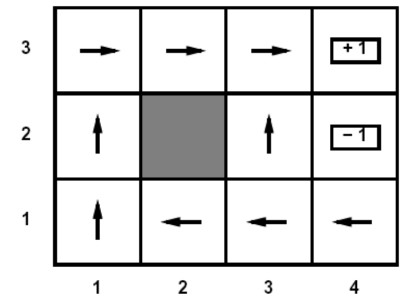
(3,3) right -1

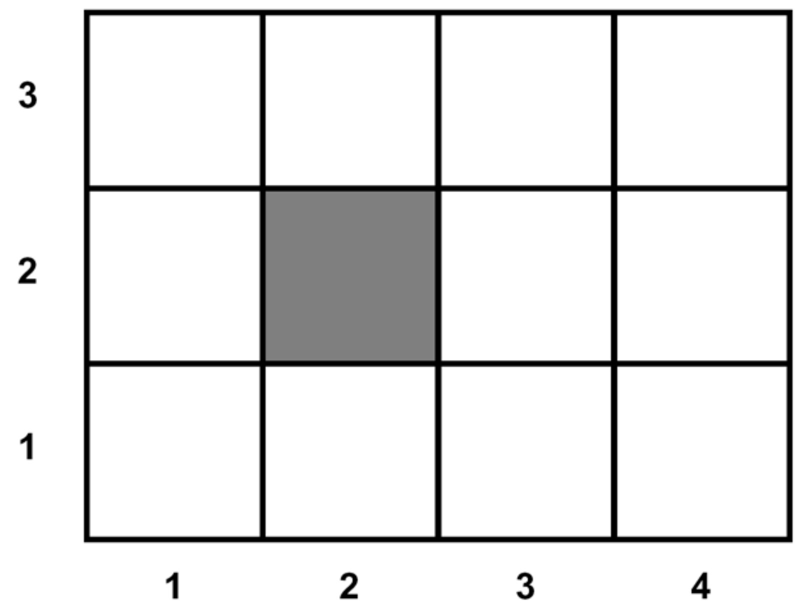
(3,2) up -1

(4,2) exit -100

(done)

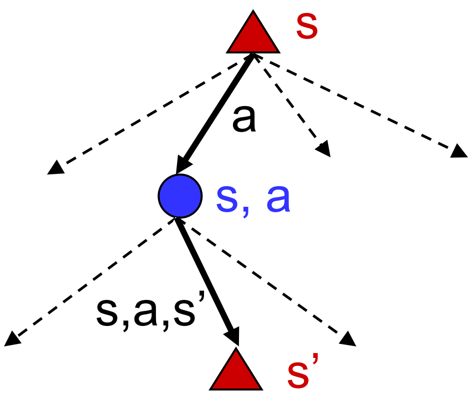
Take $\gamma = 1$, $\alpha = 0.5$





Problems with TD Value Learning

- TD value leaning is a model-free way to do policy evaluation
- However, if we want to turn values into a (new) policy, we're sunk:

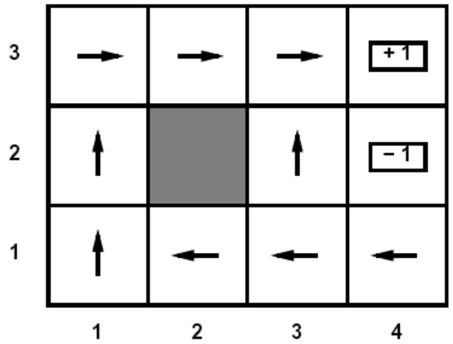


$$\pi(s) = \arg\max_a Q^*(s,a)$$

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

- Idea: learn Q-values directly
- Makes action selection model-free too!

Active Learning



- Full reinforcement learning
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You can choose any actions you like
  - Goal: learn the optimal policy
  - what value iteration did!

- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens

Q-Learning

- Q-Learning: sample-based Q-value iteration
- Learn Q*(s,a) values
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

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

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

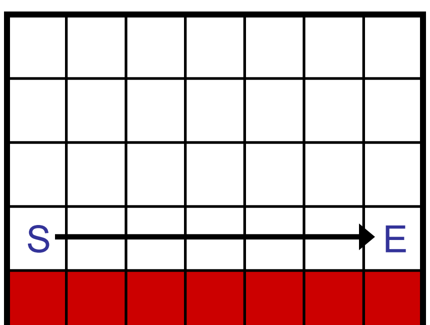
  - Incorporate the new estimate into a running average:

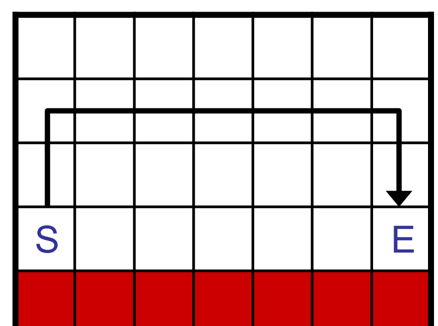
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[sample\right]$$

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy
  - If you explore enough
  - If you make the learning rate small enough
  - but not decrease it too quickly!
  - Basically doesn't matter how you select actions (!)

- Neat property: off-policy learning
  - learn optimal policy without following it (some caveats)





Exploration / Exploitation

- Several schemes for forcing exploration
  - Simplest: random actions (ε greedy)
    - Every time step, flip a coin
    - With probability ε, act randomly
    - With probability 1-ε, act according to current policy
  - Problems with random actions?
    - You do explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

Exploration Functions

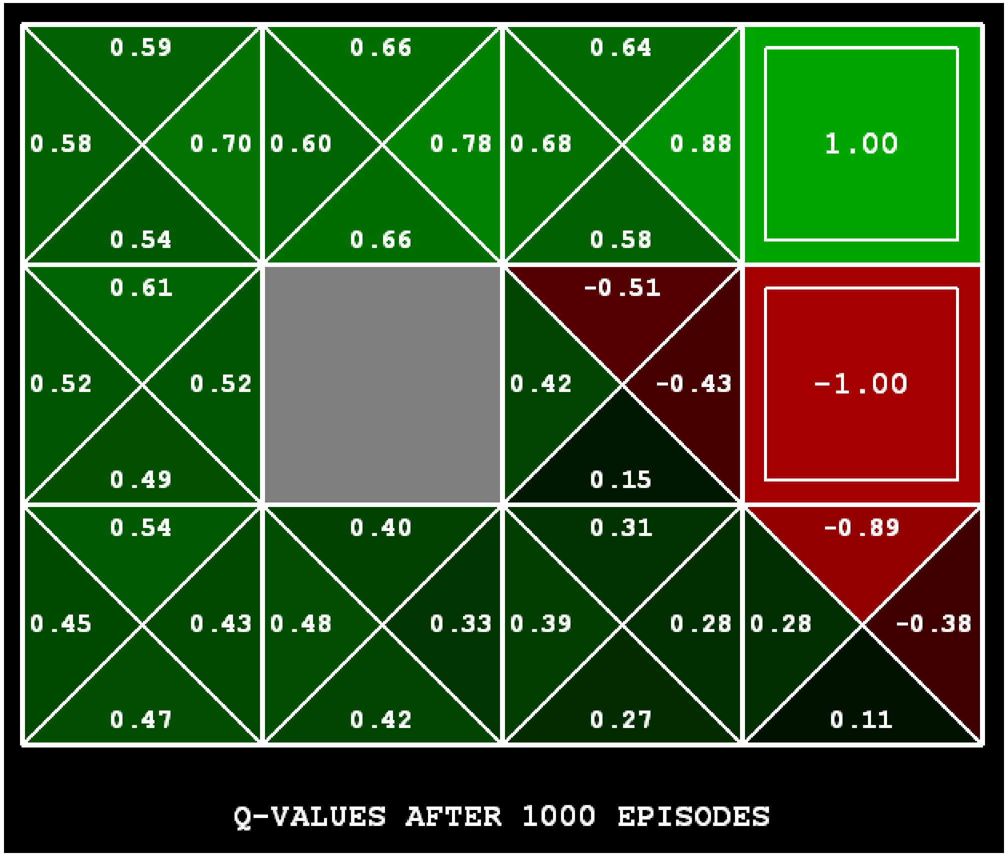
- When to explore
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established
- Exploration function
  - Takes a value estimate and a count, and returns an optimistic utility, e.g. $f(u,n) = u + k/n$ (exact form not important)

$$Q_{i+1}(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q_i(s',a')$$

$$Q_{i+1}(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q_i(s',a'), N(s',a'))$$

Q-Learning

- Q-learning produces tables of q-values:



The Story So Far: MDPs and RL

Things we know how to do:

- If we know the MDP
  - Compute V*, Q*, π* exactly
  - Evaluate a fixed policy π
- If we don't know the MDP
  - We can estimate the MDP then solve
  - We can estimate V for a fixed policy π
  - We can estimate Q*(s,a) for the optimal policy while executing an exploration policy

Techniques:

- Model-based DPs
  - Value and policy Iteration
  - Policy evaluation
- Model-based RL
- Model-free RL:
  - Value learning
  - Q-learning