Probabilistic Robotics*

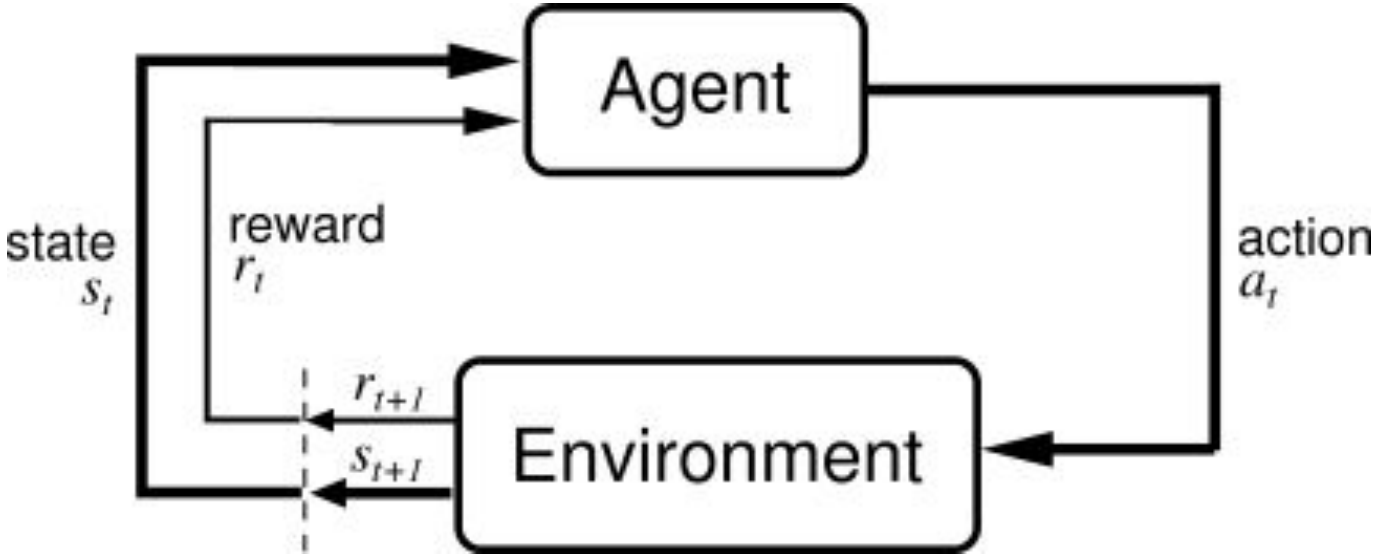
Reinforcement Learning

Mohan Sridharan
University of Birmingham, UK
m.sridharan@bham.ac.uk

*Slides adapted from Dan Klein's lectures.

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.



Reinforcement Learning

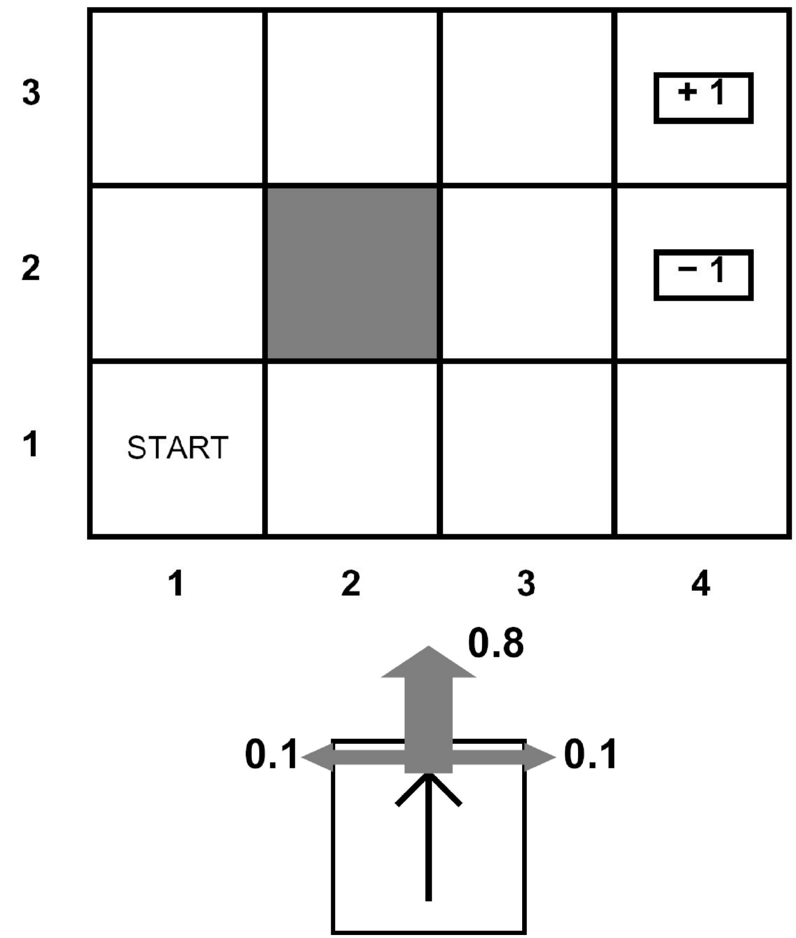
- Basic idea:
  - Receive feedback in the form of rewards.
  - Agent's utility is defined by the reward function.
  - Must learn to act so as to maximize expected rewards.
  - *Change the rewards, change the learned behavior!*

- Examples:
  - Playing a game, reward at the end for winning / losing
  - Vacuuming a house, reward for each piece of dirt picked up
  - Automated taxi, reward for each passenger delivered

- First: Need to master MDPs.

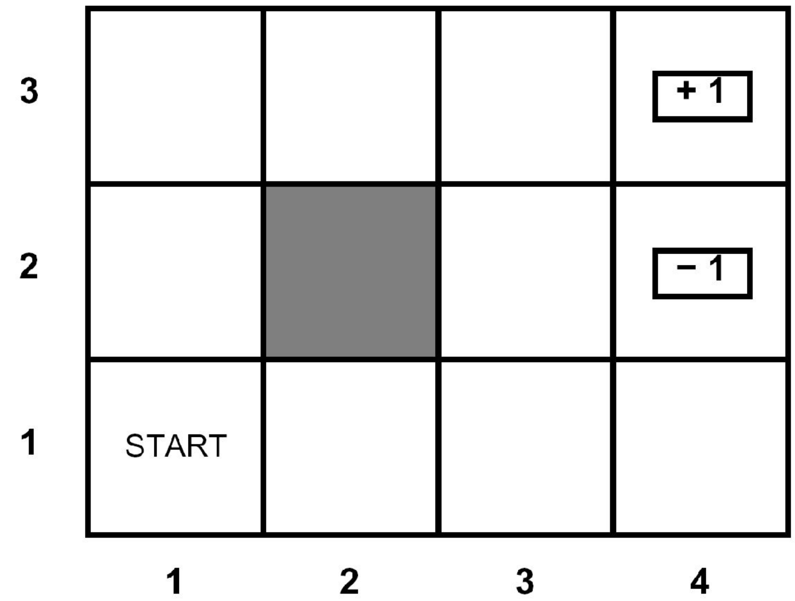
Grid World

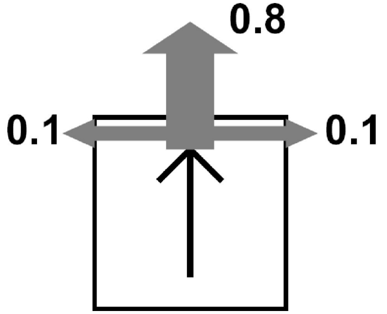
- The agent lives in a grid.
- Walls block the agent's path.
- The agent's actions do not always go as planned:
  - 80% of the time, the action North takes the agent North (if there is no wall there).
  - 10% of the time, North takes the agent West; 10% East.
  - If there is a wall in the direction the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



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'.
    - 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 the T and R are unknown.

What is Markov about MDPs?

- Andrey Markov (1856-1922)

- "Markov" generally means that given the present state, the future and the past are conditionally independent.

- For MDPs, "Markov" means:

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

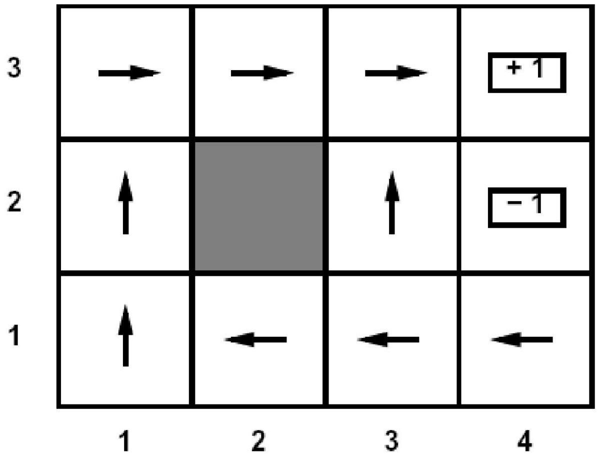
$$= P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

Solving MDPs

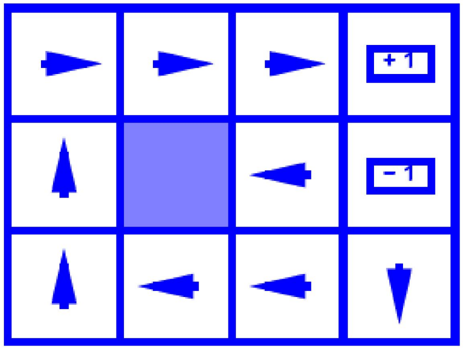
- In deterministic single-agent search problem, want an optimal plan, or sequence of actions, from start to goal.

- 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.

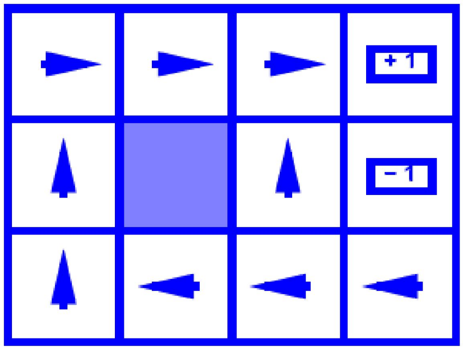
Optimal policy when R(s, a, s') = -0.03 for all non-terminal s.



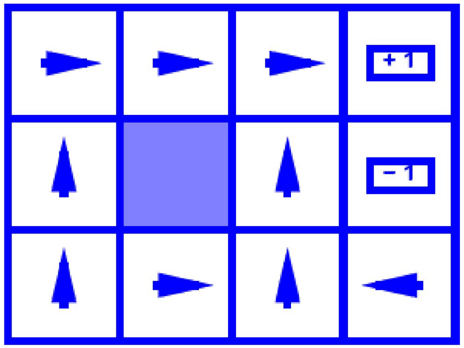
Example Optimal Policies



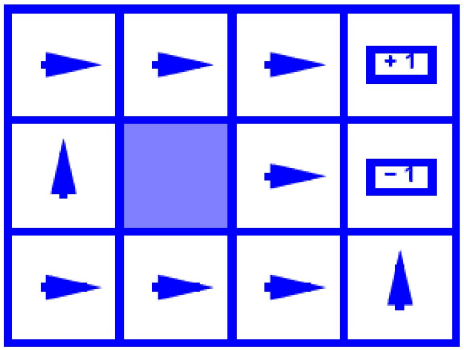
R(s) = -0.01



R(s) = -0.03



R(s) = -0.4



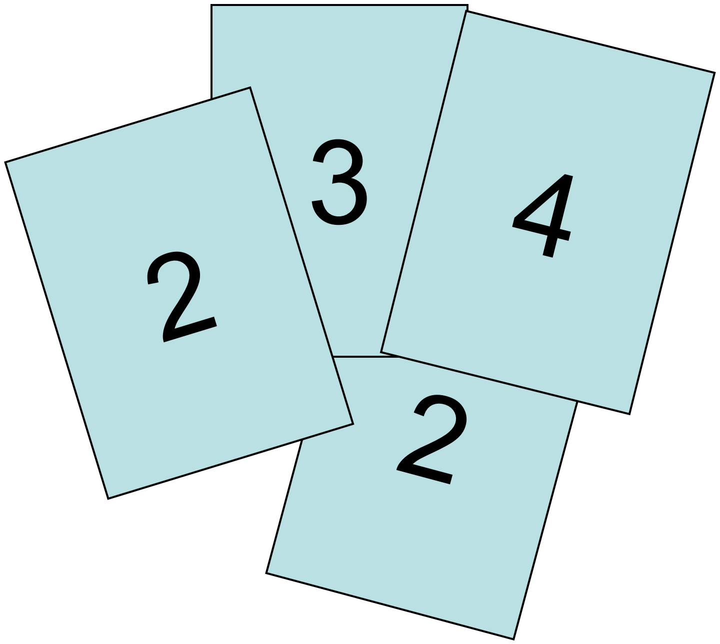
R(s) = -2.0

Example: High-Low

- Three card types: 2, 3, 4.
- Infinite deck, twice as many 2's.
- Start with 3 showing.
- Say "high" or "low" after each card.

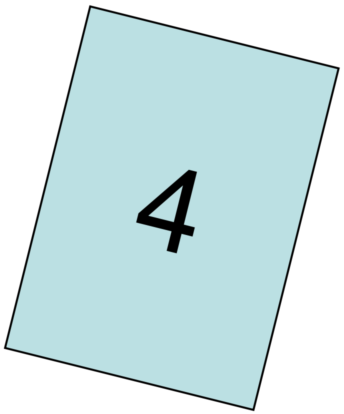
- New card is flipped:
  - If you are right, you win the points shown on the new card.
  - If you are wrong, game ends.
  - Ties are no-ops.

- Some key features:
  - #1: get rewards as you go.
  - #2: you might play forever!



High-Low

- States: 2, 3, 4, done. Start: 3.
- Actions: High, Low.
- Model: T(s, a, s'):
  - P(s'=done | 4, High) = 3/4
  - P(s'=2 | 4, High) = 0
  - P(s'=3 | 4, High) = 0
  - P(s'=4 | 4, High) = 1/4
  - P(s'=done | 4, Low) = 0
  - P(s'=2 | 4, Low) = 1/2
  - P(s'=3 | 4, Low) = 1/4
  - P(s'=4 | 4, Low) = 1/4
  - …
- Rewards: R(s, a, s'):
  - Number shown on s' if s ≠ s'.
  - 0 otherwise.



Note: could choose actions with search. How?

Utilities of Sequences

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

Temporarily assuming that reward only depends on state!

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

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

$$V([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots$$

  - Discounted utility:

$$V([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) \cdots$$

Infinite Utilities?!

- Problem: infinite sequences with infinite rewards.

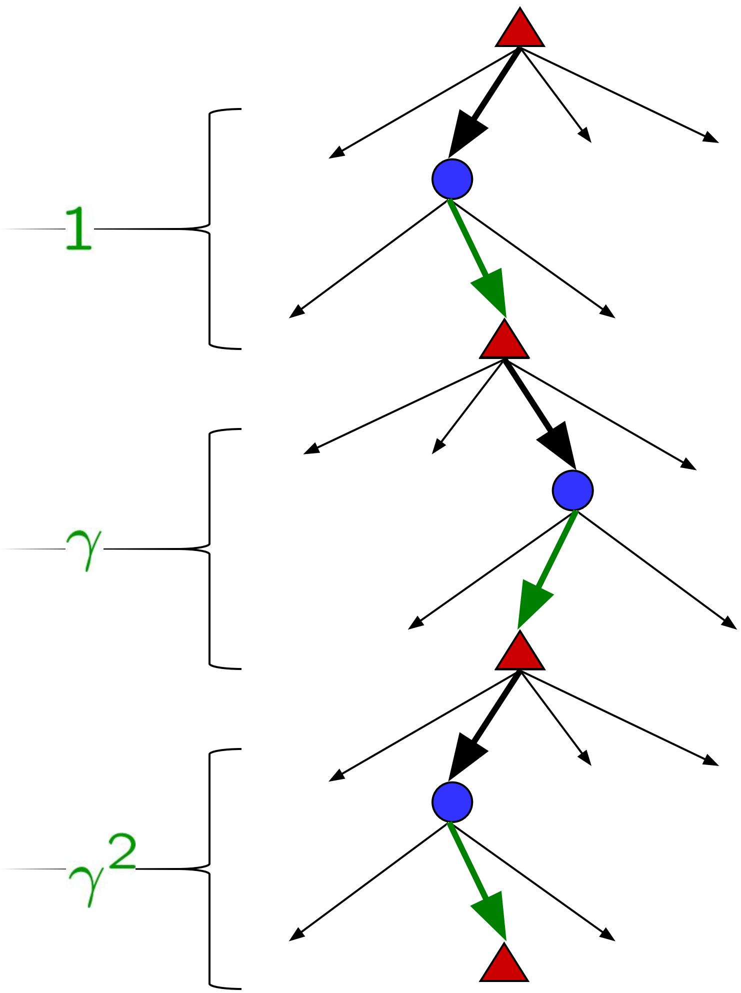
- Solutions:
  - Finite horizon:
    - Terminate after a fixed T steps.
    - Gives non-stationary policy (π depends on time left).
  - Absorbing state(s): guarantee that for every policy, agent will eventually "die" (like "done" for High-Low).
  - Discounting: for $0 < \gamma < 1$.

$$V([s_0, \ldots s_\infty]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq R_{\text{max}}/(1-\gamma)$$

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

Discounting

- Typically discount rewards by $\gamma < 1$ in each time step:
  - Rewards that come sooner have higher utility than rewards that come later.
  - Also helps the algorithms converge!



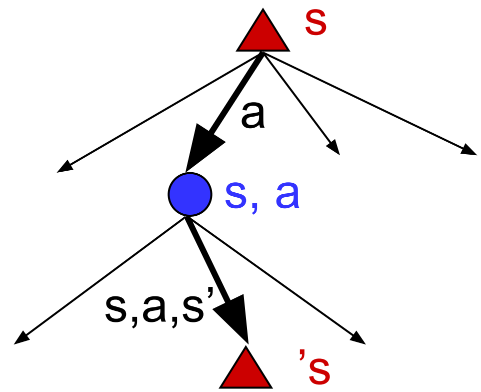
Optimal Utilities

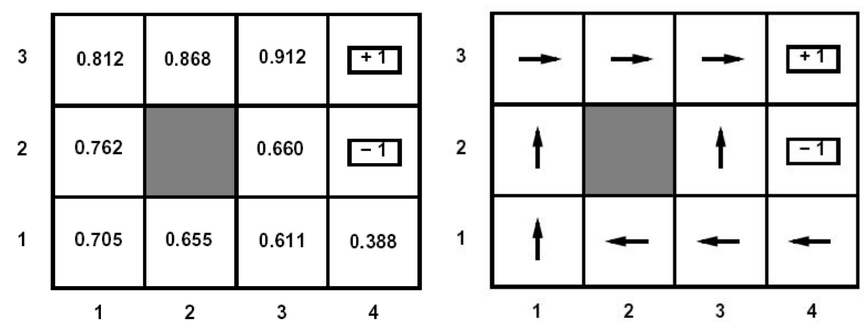
- Fundamental operation: compute the optimal utilities of states s.

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

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

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





Optimal Policies and Utilities

- Expected utility with executing **π** starting in s:

$$U^{\pi}(s) = V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R(S_t)\right]$$

- Optimal policy: $\pi_s^* = \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s)$

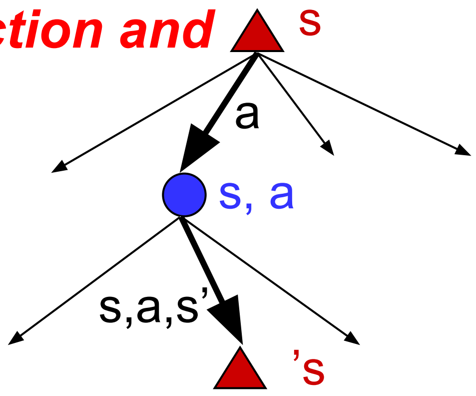
- One-step: choose action to maximize expected utility of next state:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) V(s')$$

The Bellman Equations

- Definition of “optimal utility” leads to a simple one-step look-ahead relationship amongst optimal utility values:

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



- Formal definition of *optimal* functions:

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

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

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

- *How to choose actions? how to compute optimal policy?*

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?

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

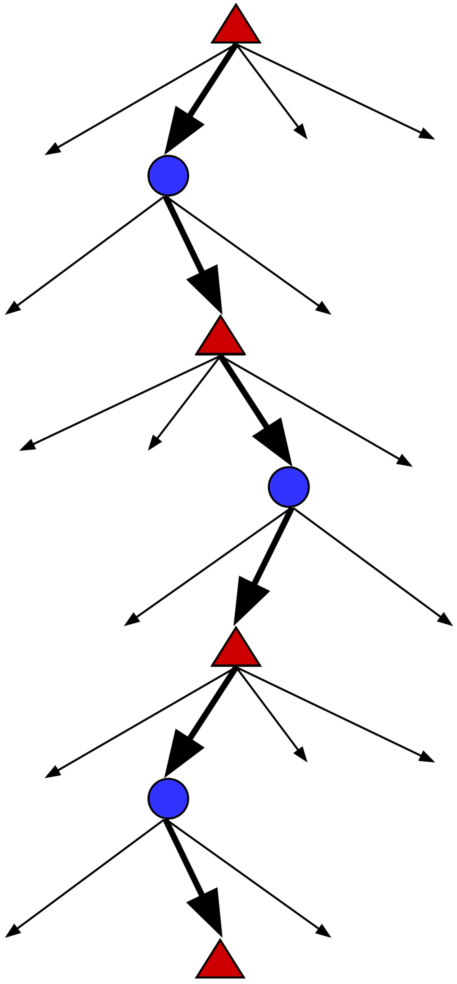
  - Lesson: actions are easier to select from Q’s!

Why Not Search Trees?

- Why not just solve search tree?

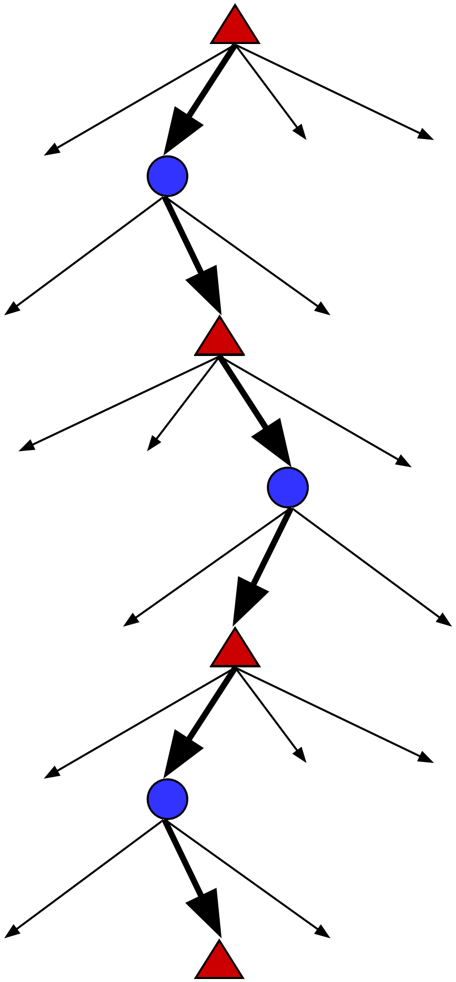
- Problems:
  - This tree is usually infinite (why?).
  - Same states appear over and over (why?).
  - We search once per state (why?).

- Idea: *Value iteration* ☺
  - Compute optimal values for all states all at once using successive approximations.
  - Will be a bottom-up dynamic program.
  - Do all planning offline, no re-planning!



Value Estimates

- Calculate estimates $V_k^*(s)$
  - *Not the optimal value of s.* Considers only next k time steps.
  - Optimal value as $k \rightarrow \infty$.
  - Why does this work?
    - With discounting, distant rewards negligible.
    - If terminal states reachable from everywhere, fraction of episodes not ending becomes negligible.
    - Otherwise, can get infinite expected utility. Then this approach will not work! ☹



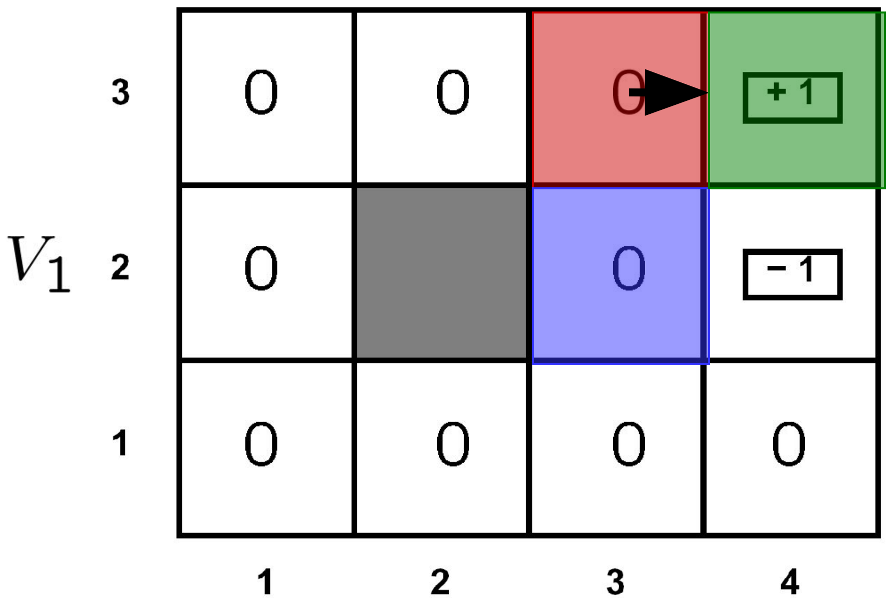
Value Iteration

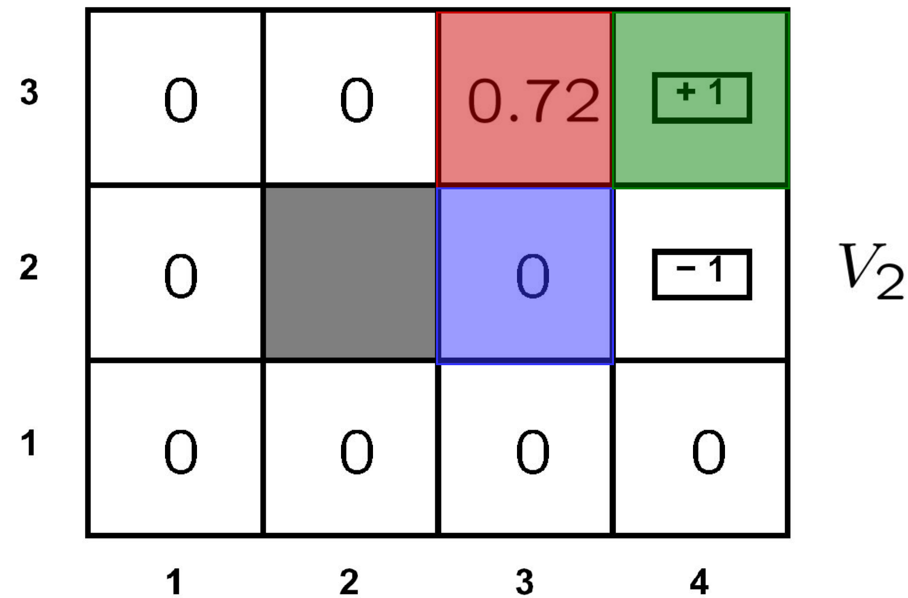
- Idea:
  - Start with $V_0(s) = 0$, which we know is right (why?)
  - Given V_i calculate the values for all states for depth i+1:

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

  - 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: 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.

Eventually: Correct Values

V_3 (when R=0, γ=*0.9*)

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

V* (when R=-.04, γ=*1*)

	1	2	3	4
3	0.812	0.868	0.918	+ 1
2	0.76		0.660	− 1
1	0.71	0.655	0.611	0.388

- This is the unique solution to the Bellman Equations!

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!
  - *How do we compute policies based on Q-values?*

Policy Evaluation

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

- **Idea 1:** turn recursive equations into updates:

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

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

- **Idea 2:** it is just a linear system, solve with Matlab (or whatever).

- Both ideas are valid solutions.

Policy Iteration

- Problem with value iteration:
  - Consider all actions in each iteration: takes |A| times longer than policy evaluation.
  - But policy does not change each iteration, i.e., time is 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 look-ahead with resulting converged (but not optimal!) utilities (slow but infrequent).
  - Repeat steps until policy converges.

- This is *policy iteration*:
  - It is 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.

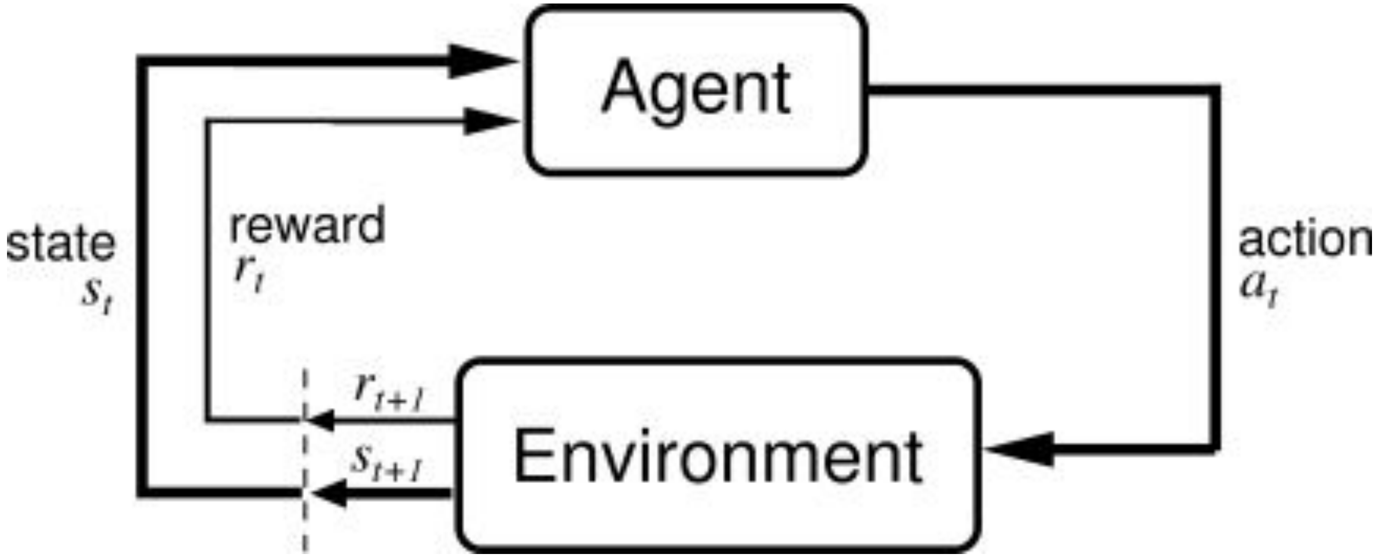
$$\pi_{k+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_k}(s') \right]$$

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.

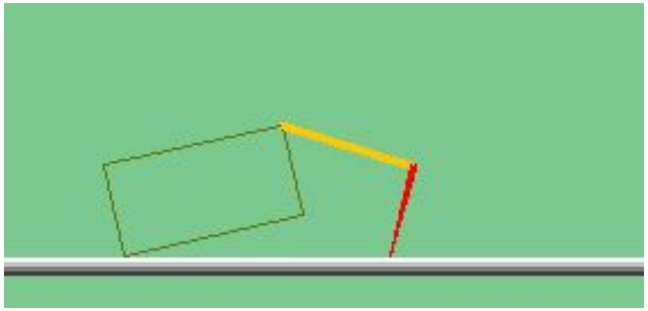
Recap: Reinforcement Learning

- 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.



Reinforcement Learning

- Reinforcement learning:
  - Still have 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.



Reinforcement Learning

Known	Unknown	Assumed
•Current state •Available actions •Experienced rewards	•Transition model •Reward structure	•Markov transitions •Fixed reward for (s,a,s')

Problem: Find optimal policy.

Model-based learning: Learn the model, solve for values.

Model-free learning: Solve for values directly (by sampling).

Three Threads of RL

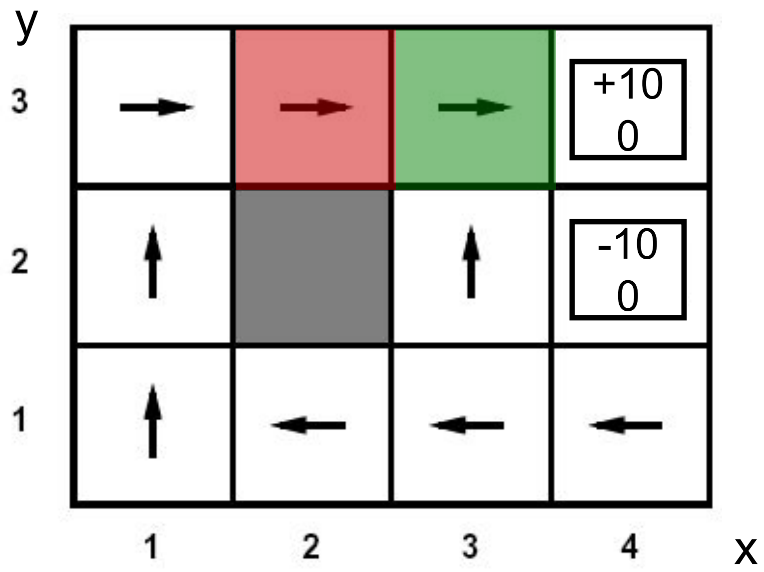
- Thread 1: Trial and error approach; origins in psychology.
- Thread 2: Dynamic programming to solve general stochastic optimal control problems; curse of dimensionality! (**Chapter 4, RL book**)
- Thread 3: temporal difference methods; driven by difference between temporally successive estimates. (**Chapter 6, RL book**)
- Common problems: credit assignment, reward specification, model design or learning.
- Consider a fixed policy first...

Example: Direct Estimation

- 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)



$\gamma = 1, R = -1$

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

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

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:

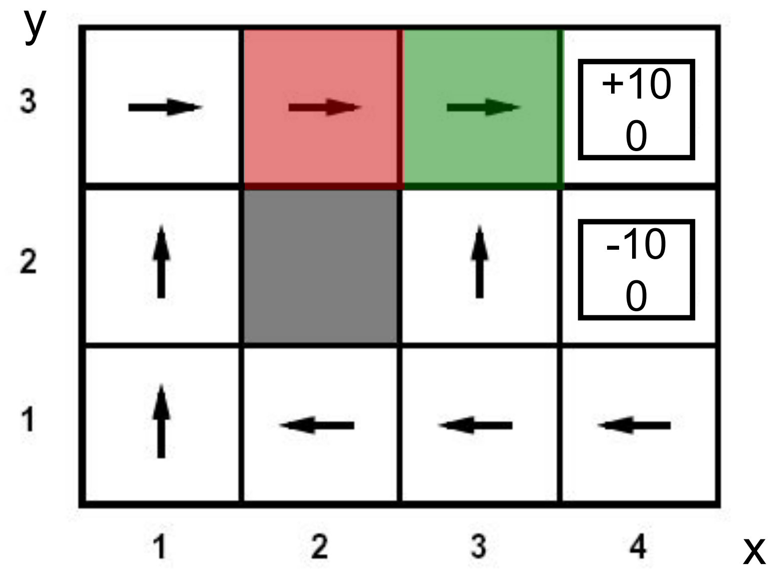
$$V^{\pi}_{i+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}_{i}(s')]$$

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.

$$x_i \sim P(x) \qquad \hat{P}(x) = \text{count}(x)/k \qquad E[f(x)] \approx \sum_x \hat{P}(x) f(x)$$

- Model-free: estimate expectation directly from samples.

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

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

Sample-Based Policy Evaluation?

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

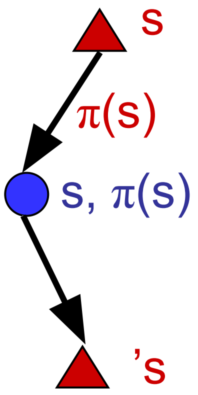
- Who needs T and R? Approximate the expectation with samples (drawn from T).

$$sample_1 = R(s,\pi(s),s_1') + \gamma V_i^{\pi}(s_1')$$
$$sample_2 = R(s,\pi(s),s_2') + \gamma V_i^{\pi}(s_2')$$
$$\ldots$$
$$sample_k = R(s,\pi(s),s_k') + \gamma V_i^{\pi}(s_k')$$

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

Temporal-Difference Learning

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



- Temporal difference learning:
  - Policy can still be 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: $$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

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

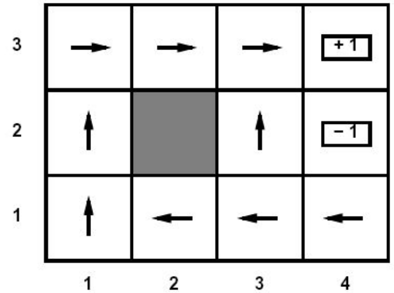
(3,3) right -1

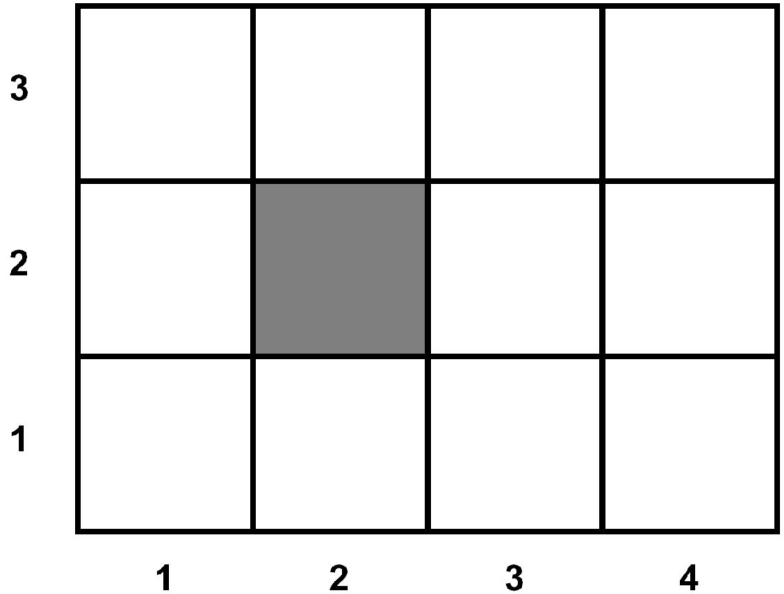
(3,2) up -1

(4,2) exit -100

(done)

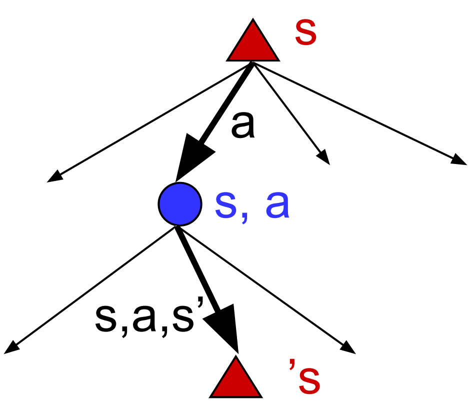
Take $\gamma = 1$, $\alpha = 0.5$





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 are 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!

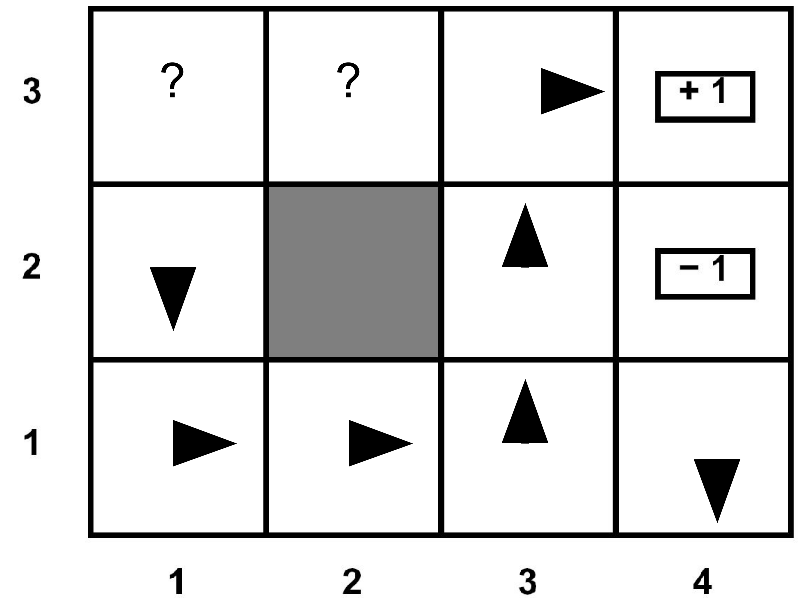
Model-Based Active Learning

- In general, want to learn the optimal policy, not evaluate a fixed policy.

- **Idea:** adaptive dynamic programming ☺
  - Learn an initial model of the environment.
  - Solve for optimal policy for this model (value or policy iteration).
  - Refine model through experience and repeat.
  - Ensure we actually learn about all of the model.

Example: Greedy ADP

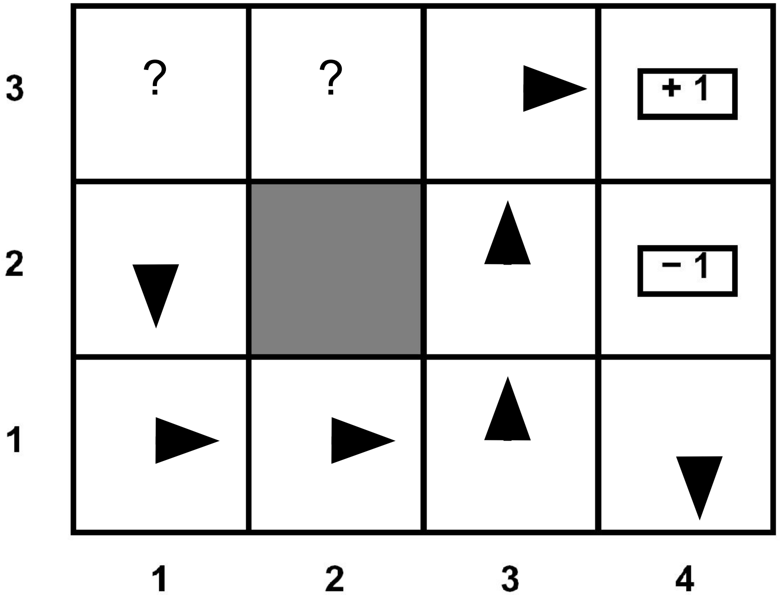
- Imagine we find the lower path to the good exit first.
- Some states will never be visited following this policy from (1,1).
- Can keep re-using this policy but following it never explores the regions of the model we need in order to learn the optimal policy .



What Went Wrong?

- Problem with following optimal policy for current model:
  - Never learns about better regions of space if current policy neglects them.

- Fundamental tradeoff: exploration vs. exploitation.
  - Exploration: take actions with suboptimal estimates to discover new rewards and increase eventual utility.
  - Exploitation: once true optimal policy is learned, exploration reduces utility.
  - *Systems must explore in the beginning and exploit in the limit. Epsilon-greedy policies.*



Detour: Q-Value Iteration

- Value iteration: find successive approx optimal values
  - 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:

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

- But Q-values are more useful!
  - Start with $Q_0^*(s,a) = 0$, which we know is right (why?)
  - Given Q_i^*, calculate the q-values for all q-states for depth i+1:

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

Q-Learning (Off-policy TD)

- We would like to do Q-value updates to each Q-state:

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

  - But cannot compute this update without knowing T, R.

- Instead, compute average as we go:
  - Receive a sample transition (s,a,r,s').
  - This sample suggests: $Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$
  - But we want to average over results from (s,a) (Why?)
  - So keep a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

Q-Learning Properties

- Will converge to optimal policy:
  - If you explore enough (i.e. visit each q-state many times).
  - If you make the learning rate small enough.
  - Basically does not matter how you select actions!
- On-policy methods: attempt to improve or evaluate policy used to make decisions. Provide “soft” policies.
- Off-policy methods: evaluate or improve a policy different from that used to make decisions.
- *On-policy vs. off-policy: Chapter 5 on RL textbook*.

Q-Learning

(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 $1-\varepsilon$, act according to current policy.

- Regret: expected gap between rewards during learning and rewards from optimal action.
  - Q-learning with random actions will converge to optimal values, but possibly very slowly, and will get low rewards on the way.
  - Results will be optimal but regret will be large.
  - How to make regret small?

Q-Learning

(Generalization and Abstraction)

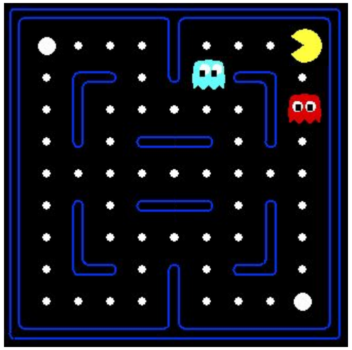
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training.
  - Too many states to hold the q-tables in memory.

- Instead, we want to generalize:
  - Learn about small number of training states from experience.
  - Generalize that experience to new, similar states.
  - *This is a fundamental idea in machine learning!*

Example: Pacman

- Let's say we discover through experience that this state is bad.

- In naïve Q-learning, we know nothing about this state or its q-states.



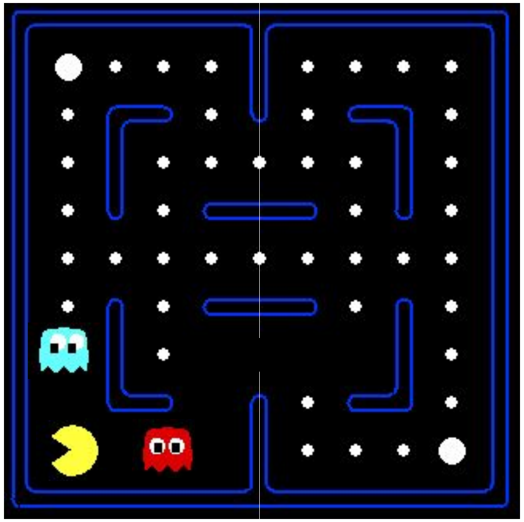
- Or even this one!

Feature-Based Representations

- **Solution:** describe a state using a vector of features (properties).
  - Features map from states to real numbers that capture important properties of the state.

  - Example features:
    - Distance to closest ghost/dot.
    - Number of ghosts.
    - 1 / (dist to dot)^2
    - Is Pacman in a tunnel? (0/1)
    - Is it the exact state on this slide?

  - Can also describe (s, a) with features (e.g. action moves closer to food).



Policy Search

- Problem: often the feature-based policies that work well are not the ones that approximate V or Q best.
  - E.g. value functions may provide horrible estimates of future rewards, but they can still produce good decisions.
  - Will see distinction between modeling and prediction again later in the course.

- **Solution:** learn the policy that maximizes rewards rather than the value that predicts rewards.

- This is the idea behind policy search, which has been used to control an upside-down helicopter!

Policy Search

- Simplest policy search:
  - Start with an initial linear value function or q-function.
  - Nudge each feature weight up and down and see if your policy is better than before.

- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical ☹

Take a Deep Breath…

- We are done MDPs and RL!

- Next: Decision-theoretic planning (POMDPs).