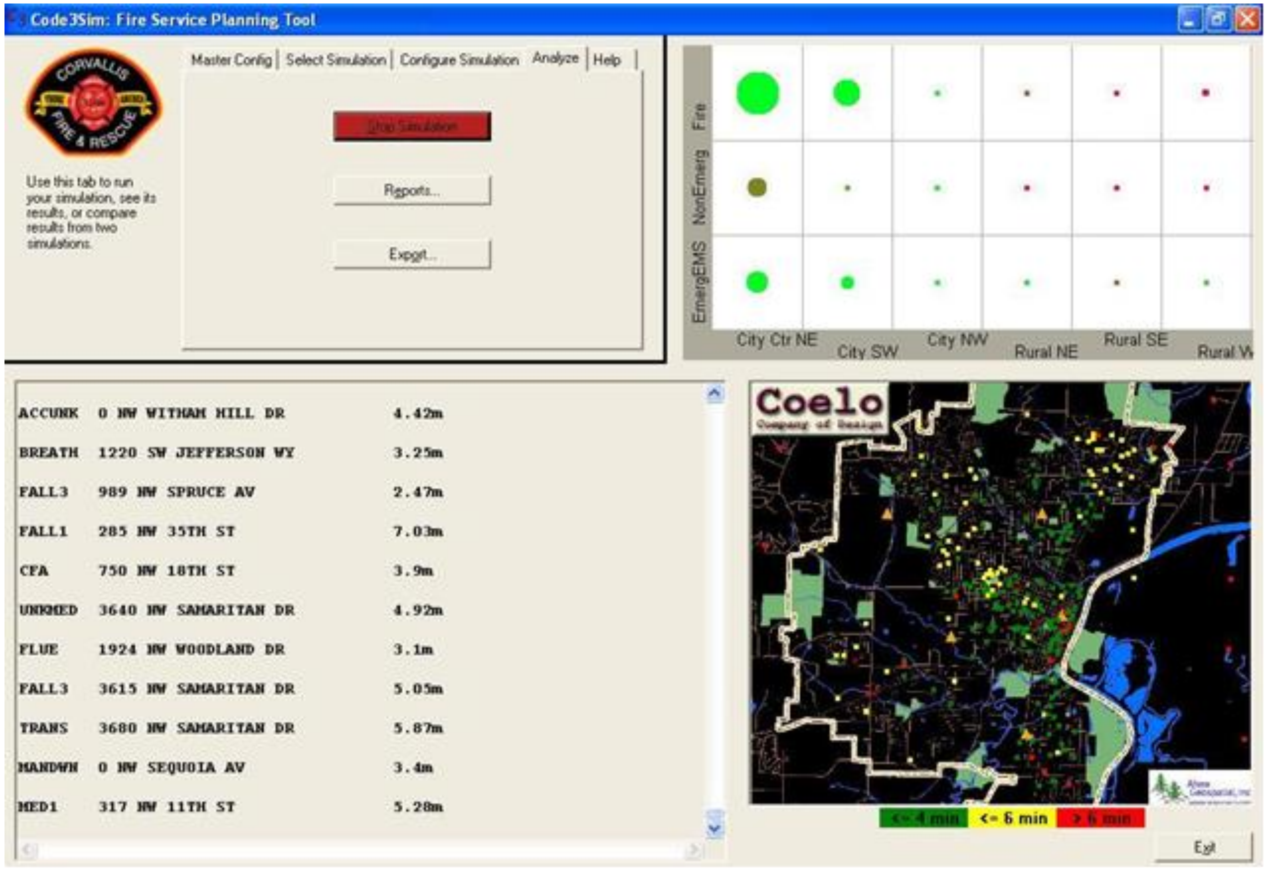
Reinforcement Learning*

* Slides based on Alan Fern

Automated Planning Under Uncertainty

Optimizing Fire & Rescue Response Policies



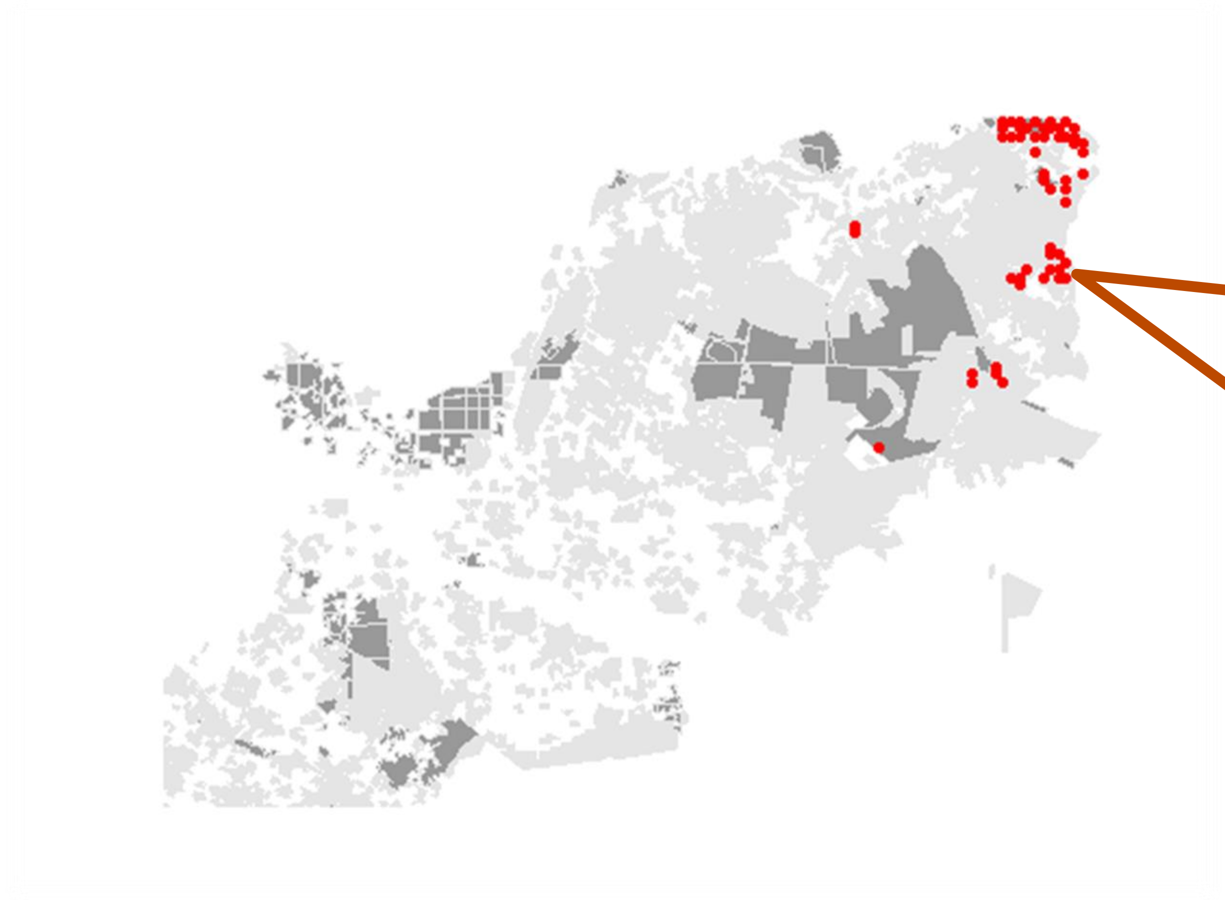
Use this tab to run your simulation, see its results, or compare results from two simulations.

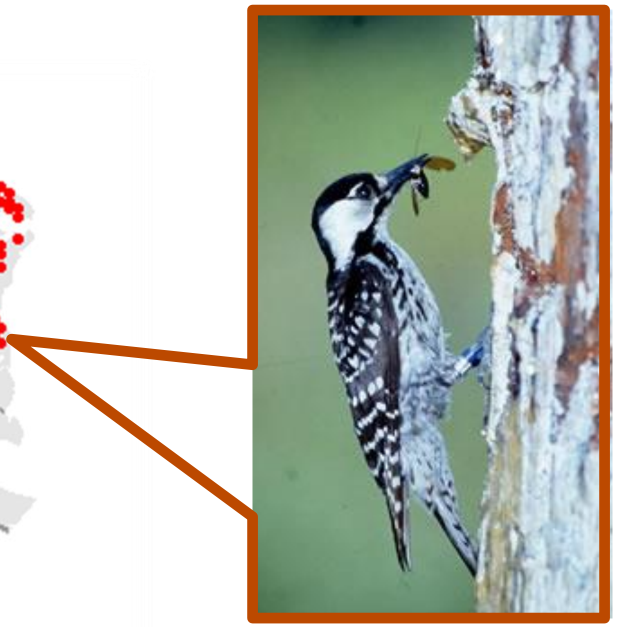
```
ACCUNK   0 NW WITHAM HILL DR        4.42m
BREATH   1220 SW JEFFERSON WY       3.25m
FALL3    989 NW SPRUCE AV           2.47m
FALL1    285 NW 35TH ST             7.03m
CFA      750 NW 18TH ST             3.9m
UNKMED   3640 NW SAMARITAN DR       4.92m
FLUE     1924 NW WOODLAND DR        3.1m
FALL3    3615 NW SAMARITAN DR       5.05m
TRANS    3680 NW SAMARITAN DR       5.87m
MANDWN   0 NW SEQUOIA AV            3.4m
MED1     317 NW 11TH ST             5.28m
```

Automated Planning Under Uncertainty

Conservation Planning:

Recovery of Red-cockaded Woodpecker

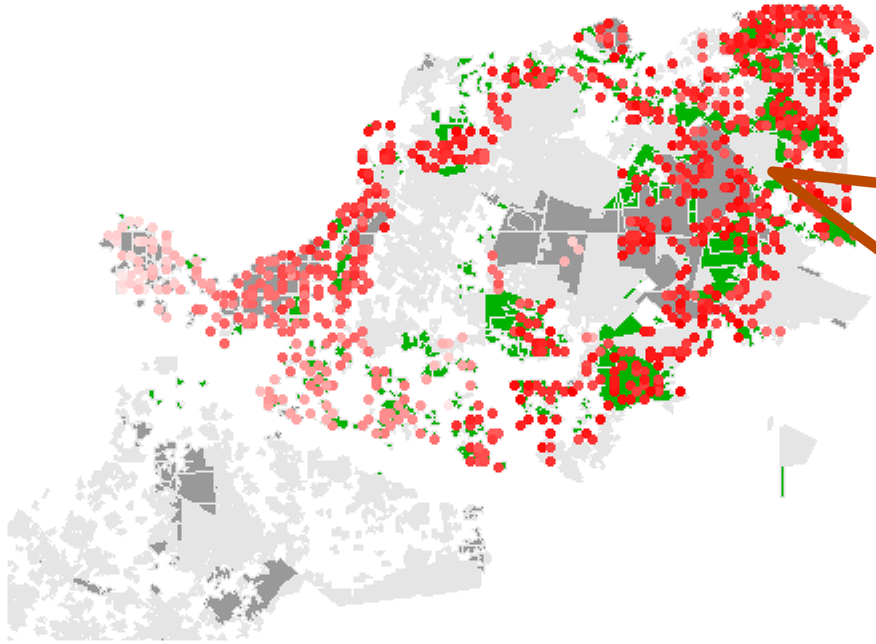




From http://www.fws.gov/rcwrecovery/rcw.html

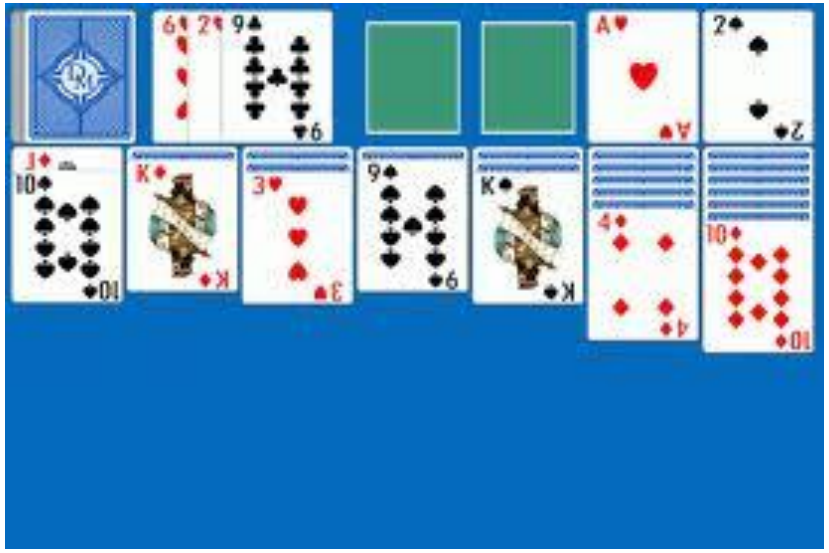
Automated Planning Under Uncertainty

Conservation Planning:
Recovery of Red-cockaded Woodpecker



From http://www.fws.gov/rcwrecovery/rcw.html

Automated Planning Under Uncertainty

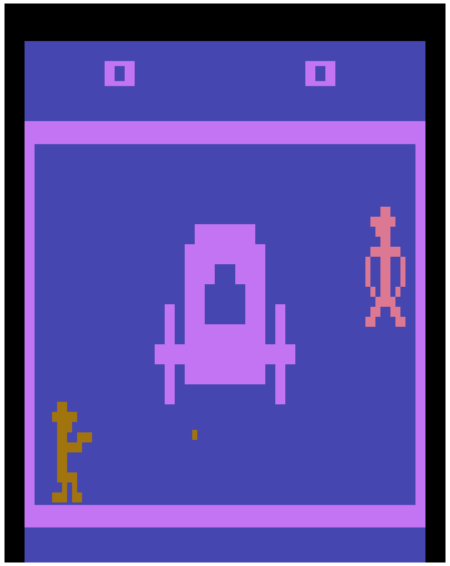


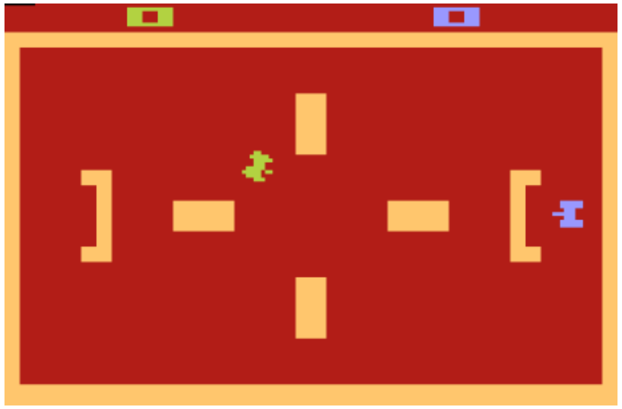
Klondike Solitaire

Farm attacked
New Footman ready
Barracks attacked
Farm attacked
Footman attacked
Farm attacked
New Peasant ready
Footman attacked
Farm attacked
Peasant attacked

Real-Time Strategy Games

AI for General Atari 2600 Games

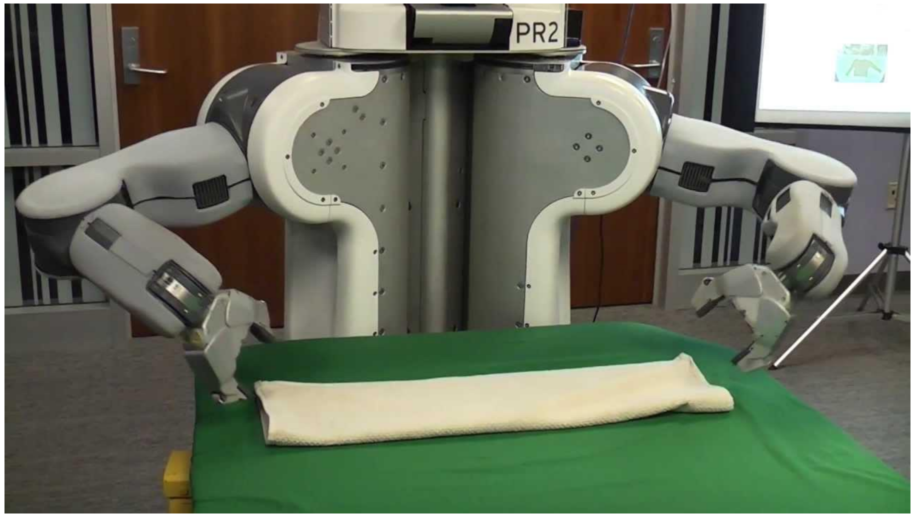




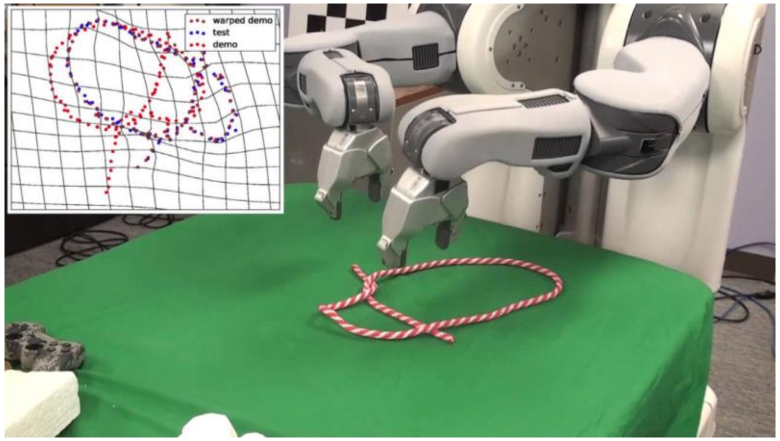
Robotics Control

Helicopter Control

Legged Robot Control



Laundry



Knot Tying

Intelligent Simulator Agents

Immersive real-time training

Smart Grids

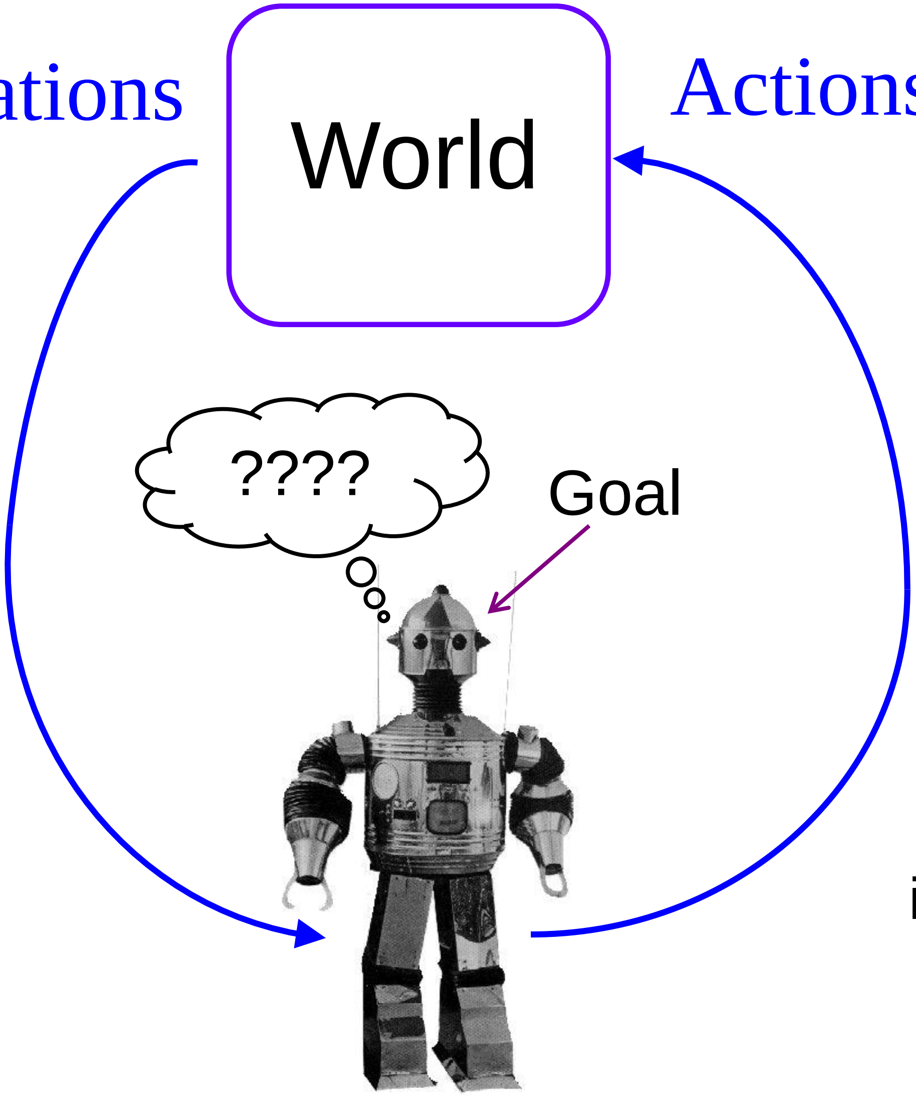
Some AI Planning Problems

- Health Care
  - Personalized treatment planning
  - Hospital Logistics/Scheduling
- Transportation
  - Autonomous Vehicles
  - Supply Chain Logistics
  - Air traffic control
- Assistive Technologies
  - Dialog Management
  - Automated assistants for elderly/disabled
  - Household robots
  - Personal planner

Common Elements

- We have a controllable system that can change state over time (in some predictable way)
  - The state describes essential information about system (the visible card information in Solitaire)
- We have an objective that specifies which states, or state sequences, are more/less preferred
- Can (partially) control the system state transitions by taking actions
- **Problem:** At each moment must select an action to optimize the overall objective
  - Produce most preferred state sequences

Some Dimensions of AI Planning



fully
observable
vs.
partially
observable

sole source
of change
vs.
other sources

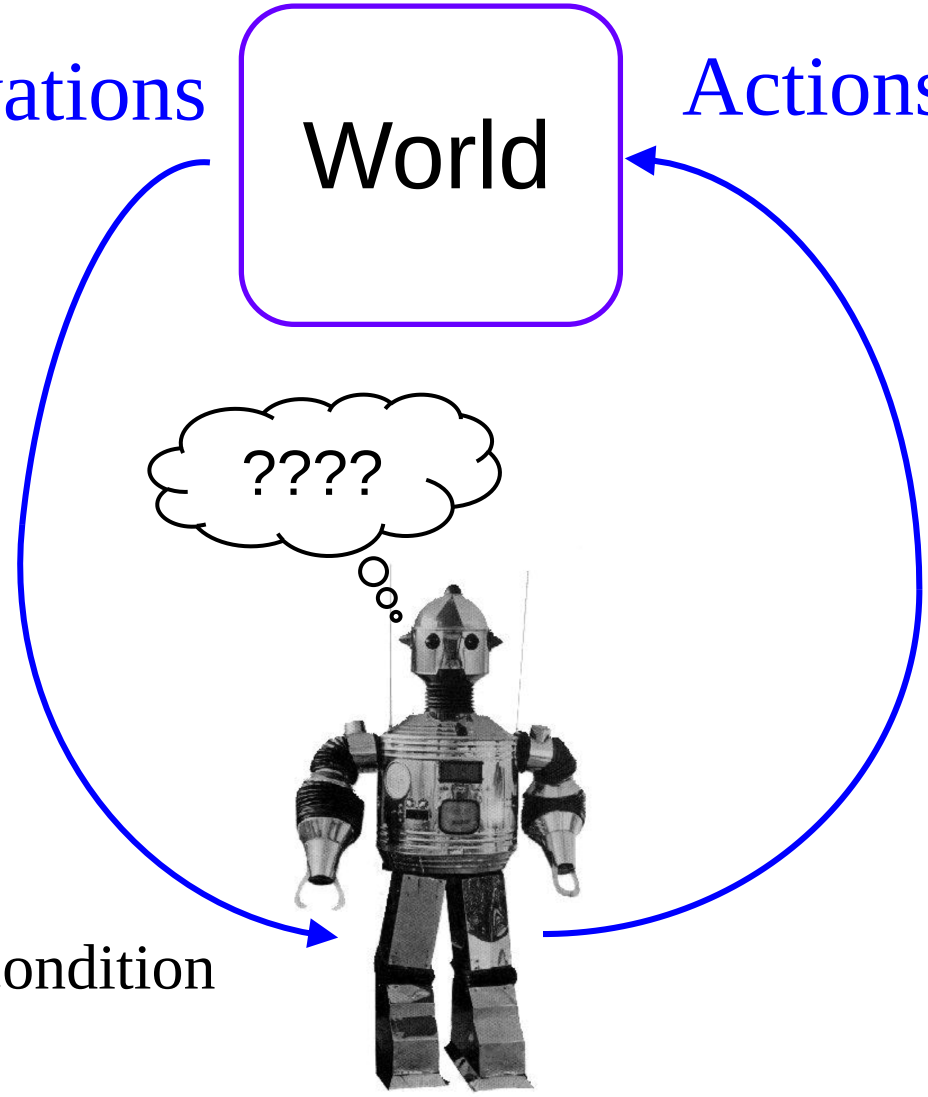
deterministic
vs.
stochastic

instantaneous
vs.
durative

Classical Planning Assumptions
(primary focus of AI planning until early 90's)

Observations

Actions



sole source
of change

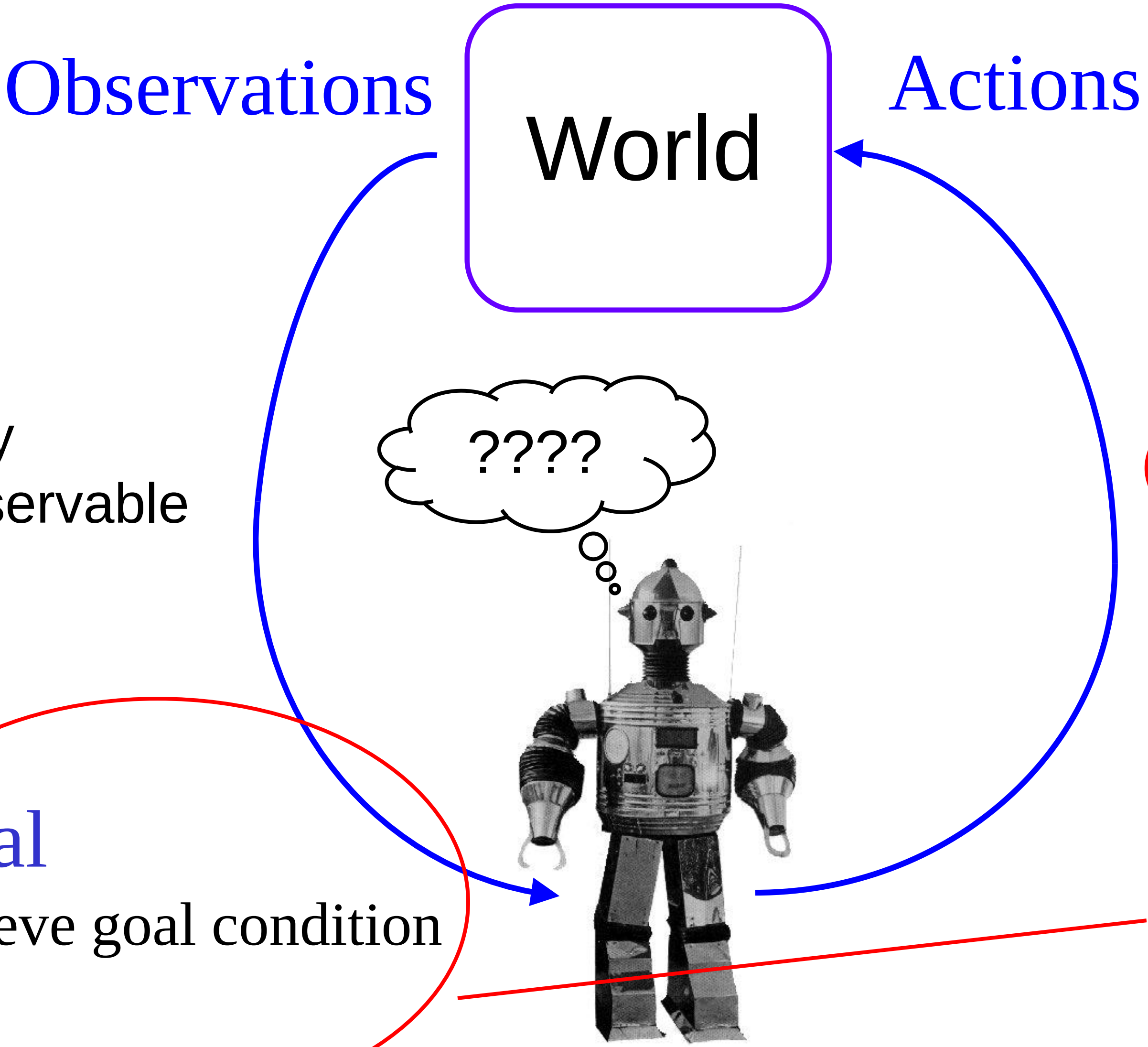
deterministic

instantaneous

fully
observable

Goal
achieve goal condition

Classical Planning Assumptions
(primary focus of AI planning until early 90's)



fully
observable

sole source
of change

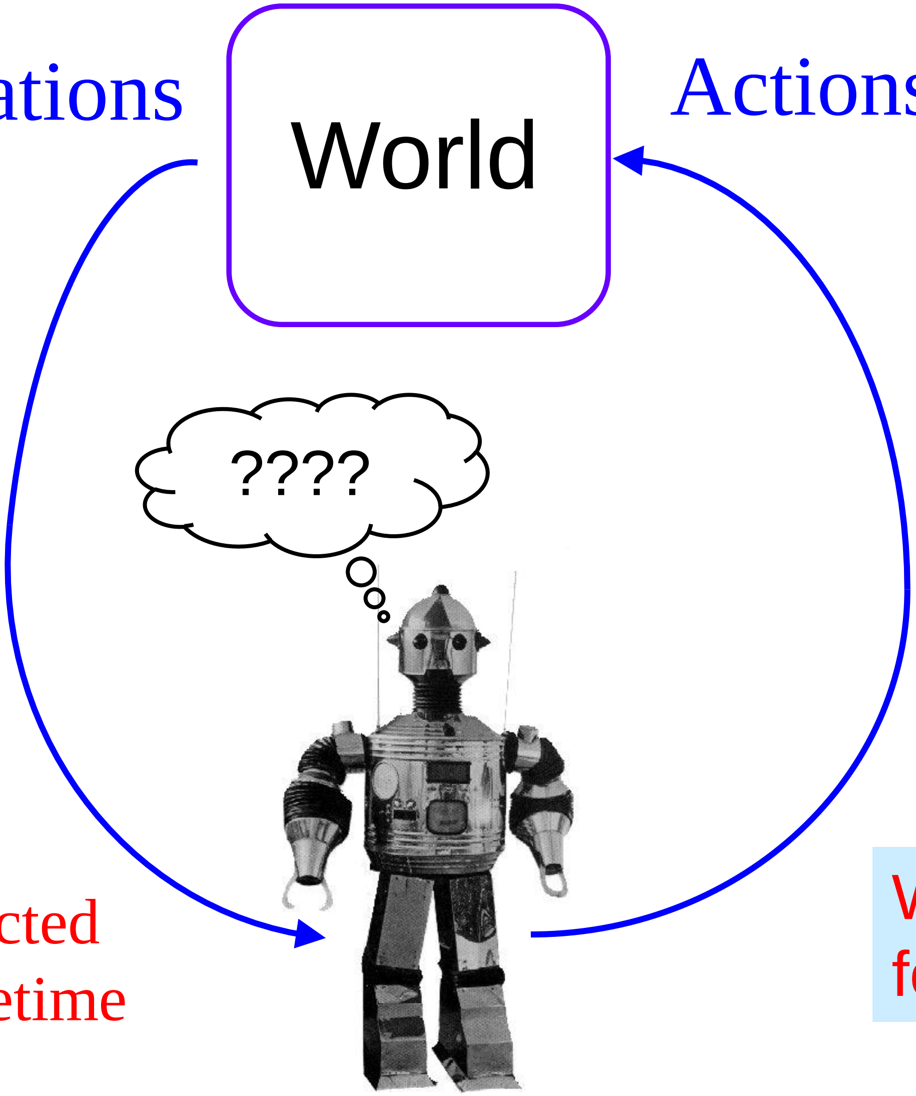
deterministic

instantaneous

Goal
achieve goal condition

Greatly limits
applicability

Stochastic/Probabilistic Planning: Markov Decision Process (MDP) Model



fully observable

sole source of change

stochastic

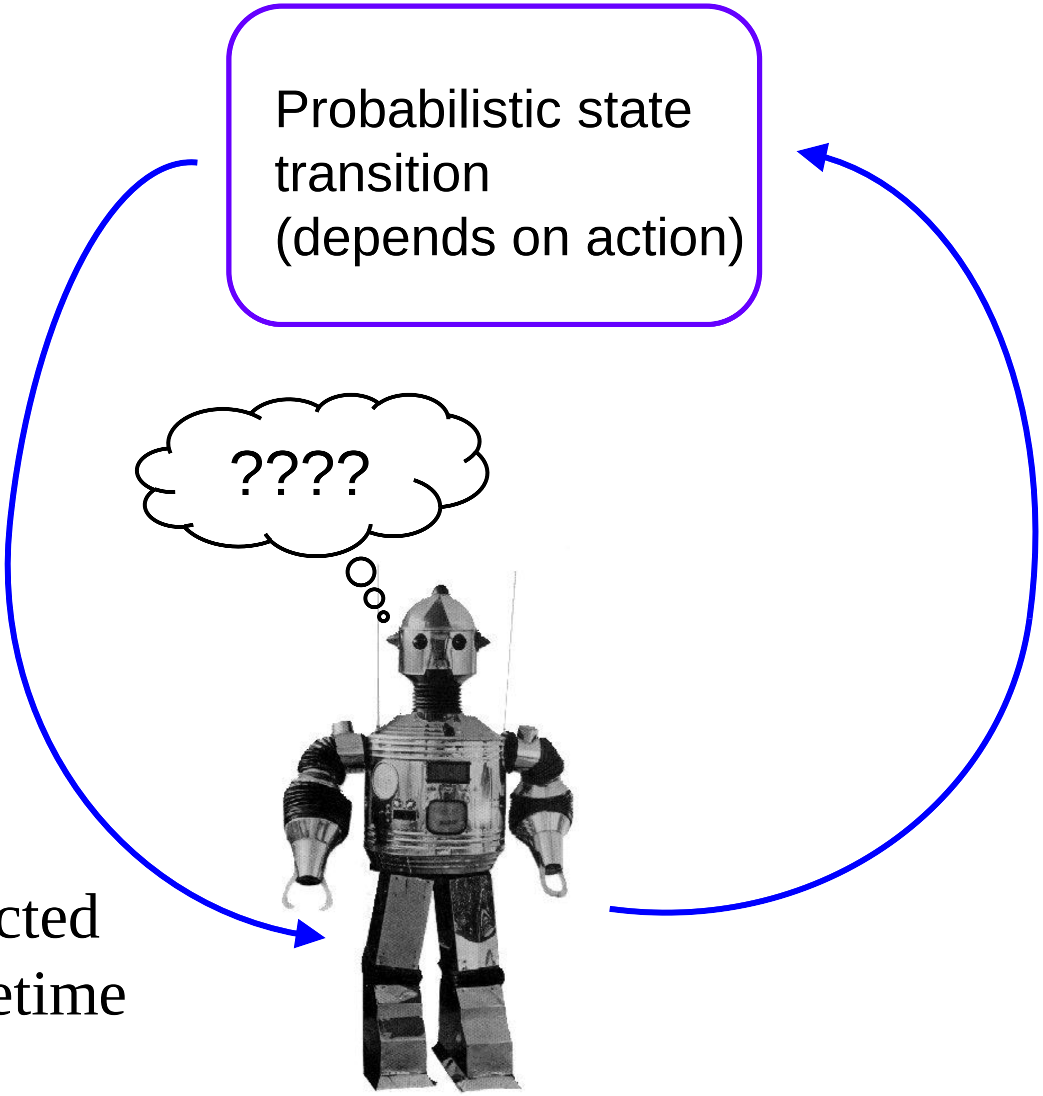
instantaneous

Goal

maximize expected reward over lifetime

We will primarily focus on MDPs

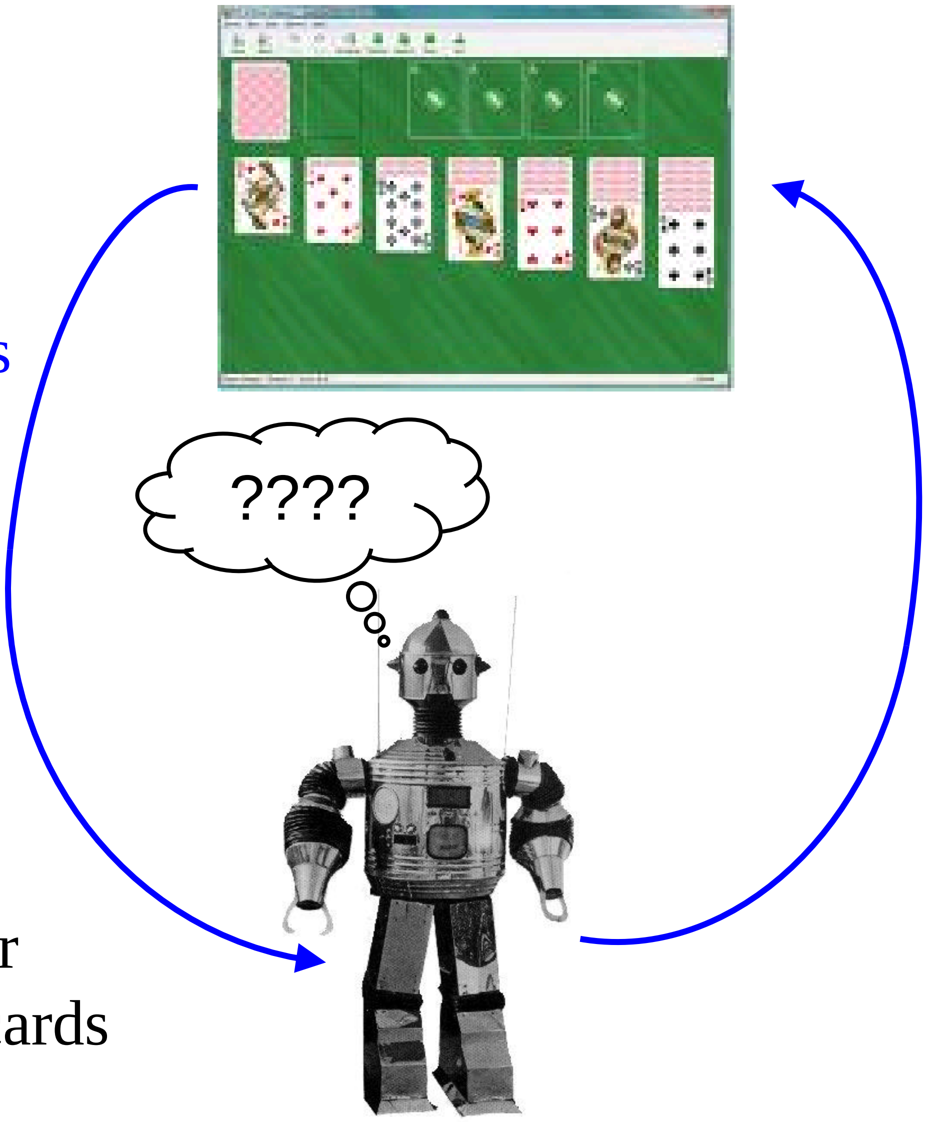
Stochastic/Probabilistic Planning: Markov Decision Process (MDP) Model



Goal

maximize expected reward over lifetime

Example MDP



State describes all visible info about cards

Action are the different legal card movements

Goal

win the game or play max # of cards

Markov Decision Processes

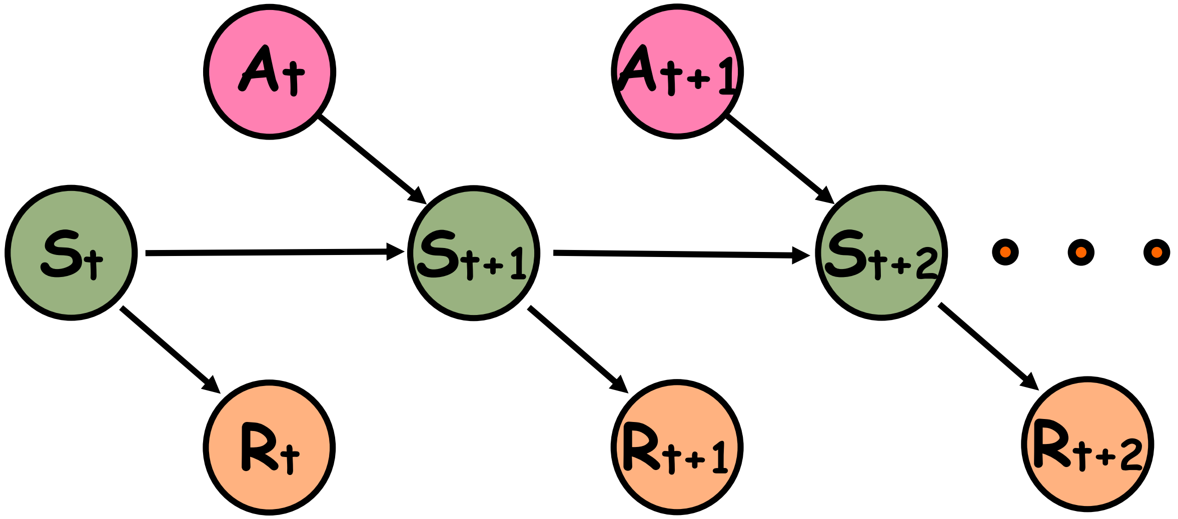
- An MDP has four components: S, A, R, T:
  - finite **state set** S (|S| = n)
  - finite **action set** A (|A| = m)
  - **transition function** T(s,a,s') = Pr(s' | s,a)
    - Probability of going to state **s'** after taking action **a** in state **s**
    - How many parameters does it take to represent?

$$m \cdot n \cdot (n-1) = O(mn^2)$$

  - bounded, real-valued **reward function** R(s)
    - Immediate reward we get for being in state s
    - Roughly speaking the objective is to select actions in order to maximize total reward
    - For example in a goal-based domain R(s) may equal 1 for reaching goal states and 0 otherwise (or -1 reward for non-goal states)

Graphical View of MDP

Dynamic Bayesian Network



Assumptions

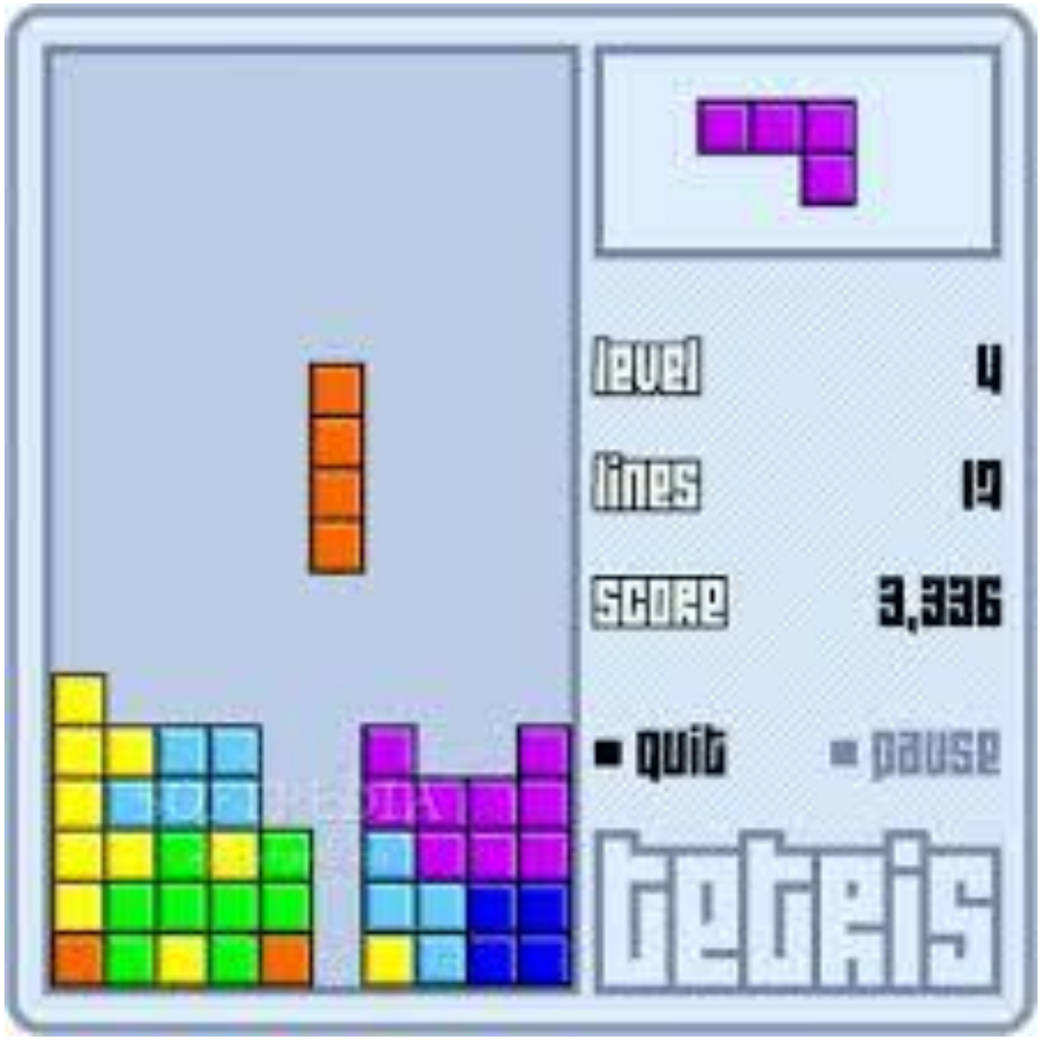
- **First-Order Markovian dynamics** (history independence)
  - $\Pr(S^{t+1}|A^t,S^t,A^{t-1},S^{t-1},..., S^0) = \Pr(S^{t+1}|A^t,S^t)$
  - Next state only depends on current state and current action

- **State-Dependent Reward**
  - $R^t = R(S^t)$
  - Reward is a deterministic function of current state and action

- **Stationary dynamics**
  - $\Pr(S^{t+1}|A^t,S^t) = \Pr(S^{k+1}|A^k,S^k)$ for all t, k
  - The world dynamics and reward function do not depend on absolute time

- **Full observability**
  - Though we can’t predict exactly which state we will reach when we execute an action, after the action is executed, we know the new state

Define an MDP that represents the game of Tetris.



A = ?

S = ?

T(s,a,s') = ?

R(s) = ?

What is a solution to an MDP?

MDP Planning Problem:

Input: an MDP (S,A,R,T)
Output: ????

What is a solution to an MDP?

MDP Planning Problem:

Input: an MDP (S,A,R,T)
Output: ????

- Should the solution to an MDP from an initial state be just a sequence of actions such as (a1,a2,a3, ….) ?
  - Consider a single player card game like Blackjack/Solitaire.
- No! In general an action sequence is not sufficient
  - Actions have stochastic effects, so the state we end up in is uncertain
  - This means that we might end up in states where the remainder of the action sequence doesn't apply or is a bad choice
  - A solution should tell us what the best action is for any possible situation/state that might arise

Policies ("plans" for MDPs)

- A solution to an MDP is a policy
  - Two types of policies: nonstationary and stationary

- Nonstationary policies are used when we are given a finite planning horizon H
  - I.e. we are told how many actions we will be allowed to take

- Nonstationary policies are functions from states and times to actions
  - π:S x T $\rightarrow$ A, where T is the non-negative integers
  - π(s,t) tells us what action to take at state s when there are t stages-to-go (note that we are using the convention that t represents stages/decisions to go, rather than the time step)

Policies (“plans” for MDPs)

- What if we want to continue taking actions indefinately?
  - Use stationary policies
- A Stationary policy is a mapping from states to actions
  - π:S → A
  - π(s) is action to do at state s (regardless of time)
  - specifies a continuously reactive controller
- Note that both nonstationary and stationary policies assume or have these properties:
  - full observability of the state
  - history-independence
  - deterministic action choice

What is a solution to an MDP?

MDP Planning Problem:

Input: an MDP (S,A,R,T)
Output: a policy such that ????

- We don't want to output just any policy
- We want to output a "good" policy
- One that accumulates a lot of reward

Value of a Policy

- How good is a policy π?
  - How do we measure reward "accumulated" by π?
- **Value function** $V: S \rightarrow \mathbb{R}$ associates value with each state (or each state and time for non-stationary π)
- $V_\pi(s)$ denotes value of policy π at state s
  - Depends on immediate reward, but also what you achieve subsequently by following π
  - An **optimal policy** is one that is no worse than any other policy at any state
- The goal of MDP planning is to compute an optimal

What is a solution to an MDP?

MDP Planning Problem:

Input: an MDP (S,A,R,T)
Output: a policy that achieves an “optimal value”

- This depends on how we define the value of a policy
- There are several choices and the solution algorithms depend on the choice
- We will consider two common choices
  - Finite-Horizon Value
  - Infinite Horizon Discounted Value

Finite-Horizon Value Functions

- We first consider maximizing expected total reward over a finite horizon
- Assumes the agent has H time steps to live

Finite-Horizon Value Functions

- We first consider maximizing expected total reward over a finite horizon
- Assumes the agent has H time steps to live
- To act optimally, should the agent use a stationary or non-stationary policy?
  - I.e. Should the action it takes depend on absolute time?
- Put another way:
  - If you had only one week to live would you act the same way as if you had fifty years to live?

Finite Horizon Problems

- Value (utility) depends on stage-to-go
  - hence use a nonstationary policy

- $V_\pi^k(s)$ is k-stage-to-go value function for π
  - expected total reward for executing π starting in s for k time steps

$$V_\pi^k(s) = E\,[\sum_{t=0}^{k} R^t \mid \pi, s\,]$$

$$= E\,[\sum_{t=0}^{k} R(s^t) \mid a^t = \pi(s^t, k-t), s^0 = s\,]$$

- Here R^t and s^t are random variables denoting the reward

Computational Problems

- There are two problems that we will be interested in solving

- **Policy evaluation**:
  - Given an MDP and a nonstationary policy π
  - Compute finite-horizon value function $V_{\pi}^{k}(s)$ for any k

- **Policy optimization**:
  - Given an MDP and a horizon H
  - Compute the optimal finite-horizon policy
  - We will see this is equivalent to computing optimal value function

Finite-Horizon Policy Evaluation

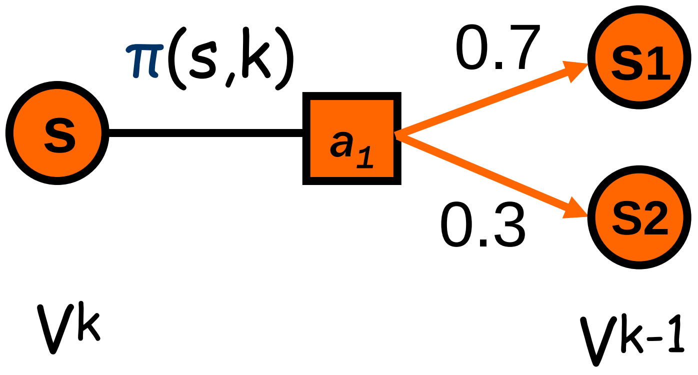
- Can use dynamic programming to compute $V_\pi^k(s)$
  - Markov property is critical for this

(k=0) $V_\pi^0(s) = R(s), \quad \forall s$

(k>0) $V_\pi^k(s) = \underbrace{R(s)}_{\text{immediate reward}} + \underbrace{\sum_{s'} T(s, \pi(s,k), s') \cdot V_\pi^{k-1}(s')}_{\text{expected future payoff with } k\text{-1 stages to go}}, \quad \forall s$

immediate reward

expected future payoff with *k*-1 stages to go



What is total time complexity?

$O(Hn^2)$

Computational Problems

- There are two problems that we will be interested in solving

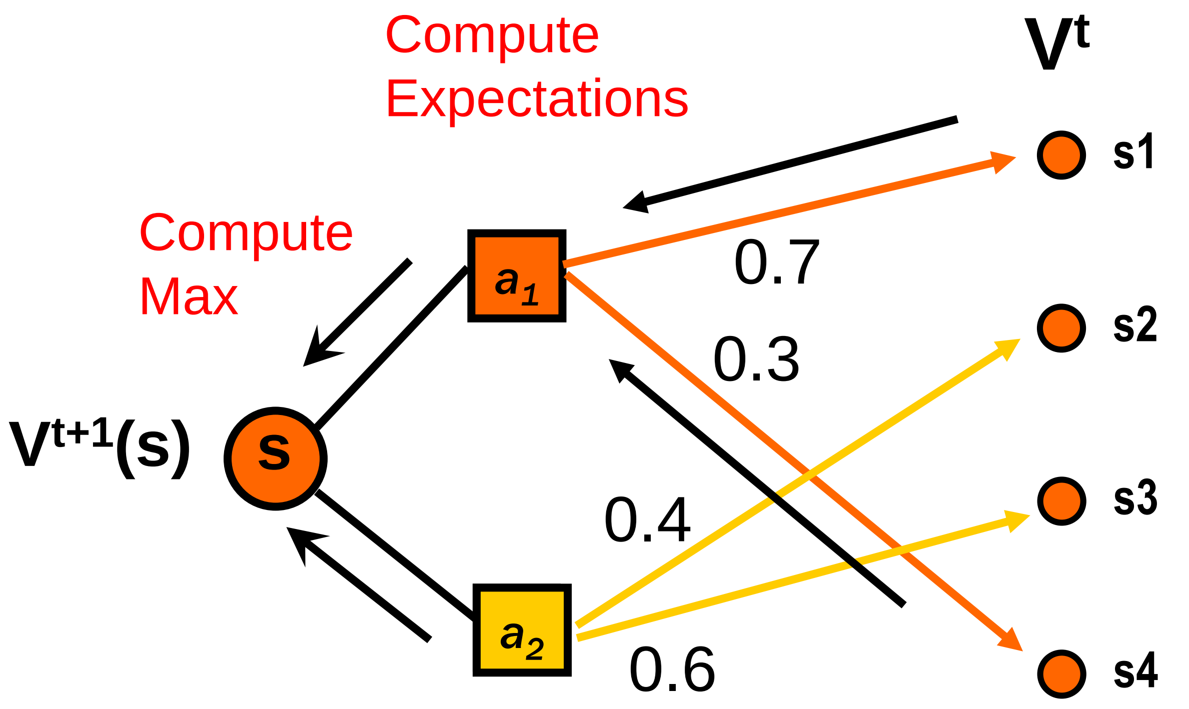
- **Policy evaluation**:
  - Given an MDP and a nonstationary policy π
  - Compute finite-horizon value function $V_\pi^k(s)$ for any k

- **Policy optimization**:
  - Given an MDP and a horizon H
  - Compute the optimal finite-horizon policy
  - We will see this is equivalent to computing optimal value function

- How many finite horizon policies are there?
  - $|A|^{Hn}$
  - So can't just enumerate policies for efficient optimization

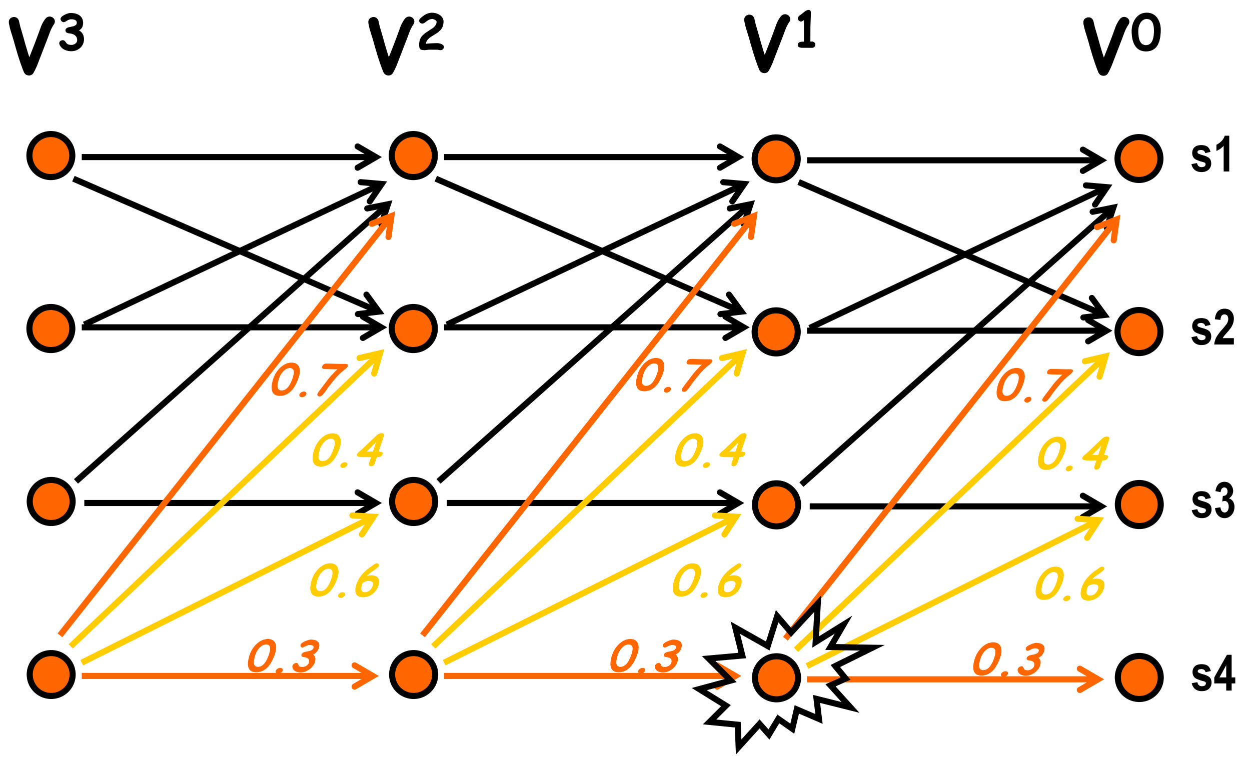
Policy Optimization: Bellman Backups

How can we compute the optimal $\mathbf{V^{t+1}(s)}$ given optimal $\mathbf{V^t}$?



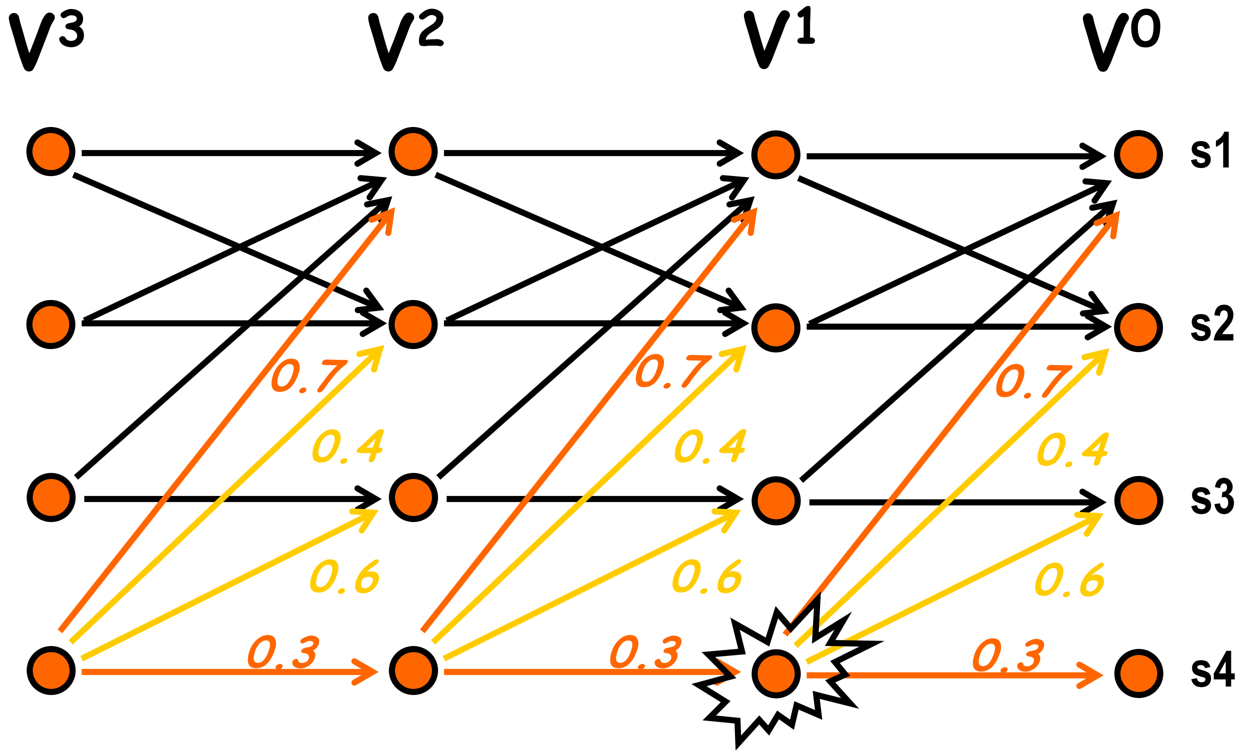
$$
\mathbf{V^{t+1}(s) = R(s) + \max \left\{ \begin{array}{ll} 0.7\, V^t(s1) + 0.3\, V^t(s4) & 🟧 \\ 0.4\, V^t(s2) + 0.6\, V^t(s3) & 🟨 \end{array} \right\}}
$$

Value Iteration



$$V^1(s4) = R(s4)+\max\{0.7\ V^0(s1) + 0.3\ V^0(s4)\quad 🟧$$
$$0.4\ V^0(s2) + 0.6\ V^0(s3)\ \}\quad 🟨$$

Value Iteration



$\Pi^*(s4,t)$ = max {🟧 🟨}

Value Iteration: Finite Horizon Case

- Markov property allows exploitation of DP principle for optimal policy construction
  - no need to enumerate $|A|^{Hn}$ possible policies
- Value Iteration

Bellman backup

$$V^0(s) = R(s), \quad \forall s$$

$$\boxed{V^k(s) = R(s) + \max_a \sum_{s'} T(s,a,s') \cdot V^{k-1}(s')}$$

$$\pi^*(s,k) = \arg\max_a \sum_{s'} T(s,a,s') \cdot V^{k-1}(s')$$

V^k is optimal k-stage-to-go value function

Π*(s,k) is optimal k-stage-to-go policy

Value Iteration: Complexity

- Note how DP is used
  - optimal soln to k-1 stage problem can be used without modification as part of optimal soln to k-stage problem

- What is the computational complexity?
  - H iterations
  - At each iteration, each of n states, computes expectation for m actions
  - Each expectation takes O(n) time
- Total time complexity: $O(Hmn^2)$
  - Polynomial in number of states. Is this good?

Summary: Finite Horizon

- Resulting policy is optimal

$$V_{\pi*}^{k}(s) \geq V_{\pi}^{k}(s), \quad \forall \pi, s, k$$

  - convince yourself of this (use induction on k)

- Note: optimal value function is unique.
- Is the optimal policy unique?
  - No. Many policies can have same value (there can be ties among actions during Bellman backups).

Discounted Infinite Horizon MDPs

- Defining value as total reward is problematic with infinite horizons (r1 + r2 + r3 + r4 + …..)
  - many or all policies have infinite expected reward
  - some MDPs are ok (e.g., zero-cost absorbing states)

- "Trick": introduce discount factor $0 \leq \beta < 1$
  - future rewards discounted by β per time step

$$V_\pi(s) = E\left[\sum_{t=0}^{\infty} \beta^t R^t \mid \pi, s\right]$$

Bounded Value

- Note: $V_\pi(s) \leq E\left[\sum_{t=0}^{\infty} \beta^t R^{\max}\right] = \dfrac{1}{1-\beta} R^{\max}$

- Motivation: economic? prob of death? convenience?

Recap: things you should know

- What is an MDP?
- What is a policy?
  - Stationary and non-stationary
- What is a value function?
  - Finite-horizon and infinite horizon
- How to evaluate policies?
  - Finite-horizon
  - Time/space complexity?
- How to optimize policies?
  - Finite-horizon
  - Time/space complexity?
  - Why they are correct?

So far ….

- Given an MDP model we know how to find optimal policies (for moderately-sized MDPs)
  - Value Iteration or Policy Iteration
- Given just a simulator of an MDP we know how to select actions
  - Monte-Carlo Planning
- What if we don’t have a model or simulator?
  - Like when we were babies . . .
  - Like in many real-world applications
  - All we can do is wander around the world observing what happens, getting rewarded and punished
- Enters reinforcement learning

Reinforcement Learning

- No knowledge of environment
  - Can only act in the world and observe states and reward
- Many factors make RL difficult:
  - Actions have non-deterministic effects
    - Which are initially unknown
  - Rewards / punishments are infrequent
    - Often at the end of long sequences of actions
    - How do we determine what action(s) were really responsible for reward or punishment? (credit assignment)
  - World is large and complex
- Nevertheless learner must decide what actions to take
  - We will assume the world behaves as an MDP

Pure Reinforcement Learning vs. Monte-Carlo Planning

- In pure reinforcement learning:
  - the agent begins with no knowledge
  - wanders around the world observing outcomes
- In Monte-Carlo planning
  - the agent begins with no declarative knowledge of the world
  - has an interface to a world simulator that allows observing the outcome of taking any action in any state

- The simulator gives the agent the ability to “teleport” to any state, at any time, and then apply any action
- A pure RL agent does not have the ability to teleport
  - Can only observe the outcomes that it happens to reach

Pure Reinforcement Learning vs. Monte-Carlo Planning

- MC planning is sometimes called RL with a “strong simulator”
  - I.e. a simulator where we can set the current state to any state at any moment
- Pure RL is sometimes called RL with a “weak simulator”
  - I.e. a simulator where we cannot set the state
- A strong simulator can emulate a weak simulator
  - So pure RL can be used in the MC planning framework
  - But not vice versa

Passive vs. Active learning

- Passive learning
  - The agent has a fixed policy and tries to learn the utilities of states by observing the world go by
  - Analogous to policy evaluation
  - Often serves as a component of active learning algorithms
  - Often inspires active learning algorithms
- Active learning
  - The agent attempts to find an optimal (or at least good) policy by acting in the world
  - Analogous to solving the underlying MDP, but without first being given the MDP model

Model-Based vs. Model-Free RL

- *Model based approach to RL:*
  - learn the MDP model, or an approximation of it
  - use it for policy evaluation or to find the optimal policy

- *Model free approach to RL:*
  - derive the optimal policy without explicitly learning the model
  - useful when model is difficult to represent and/or learn

- We will consider both types of approaches

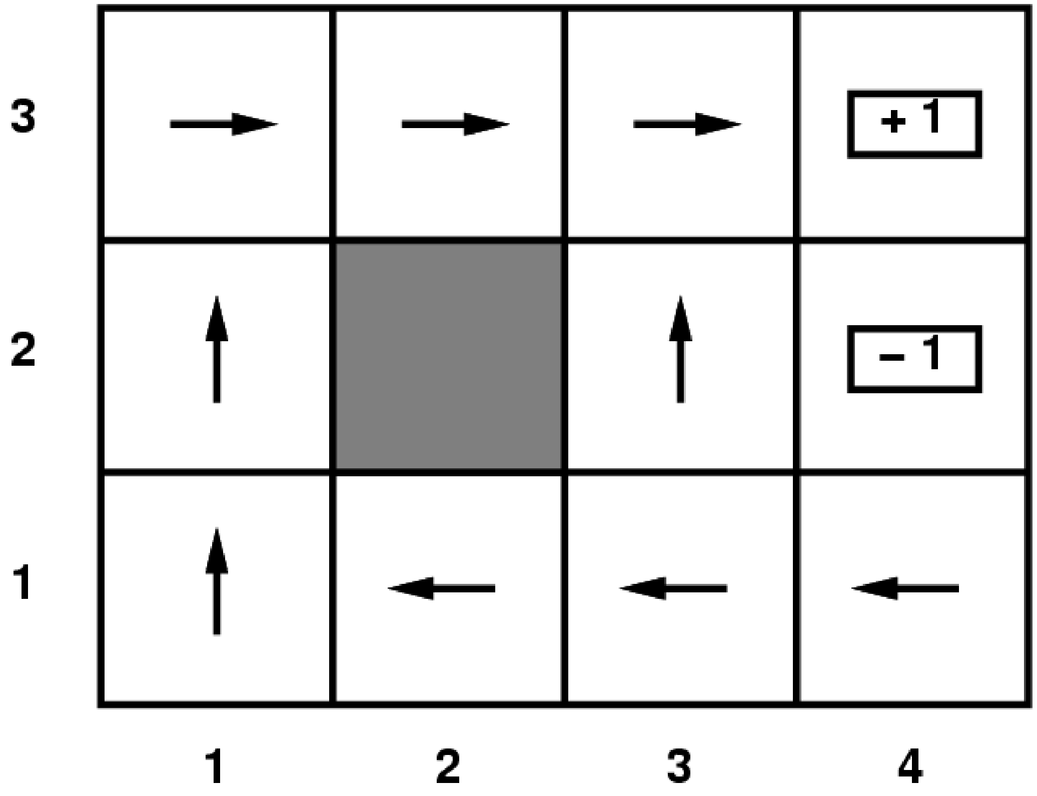
Small vs. Huge MDPs

- We will first cover RL methods for small MDPs
  - MDPs where the number of states and actions is reasonably small
  - These algorithms will inspire more advanced methods

- Later we will cover algorithms for huge MDPs
  - Function Approximation Methods
  - Policy Gradient Methods
  - Least-Squares Policy Iteration

Example: Passive RL

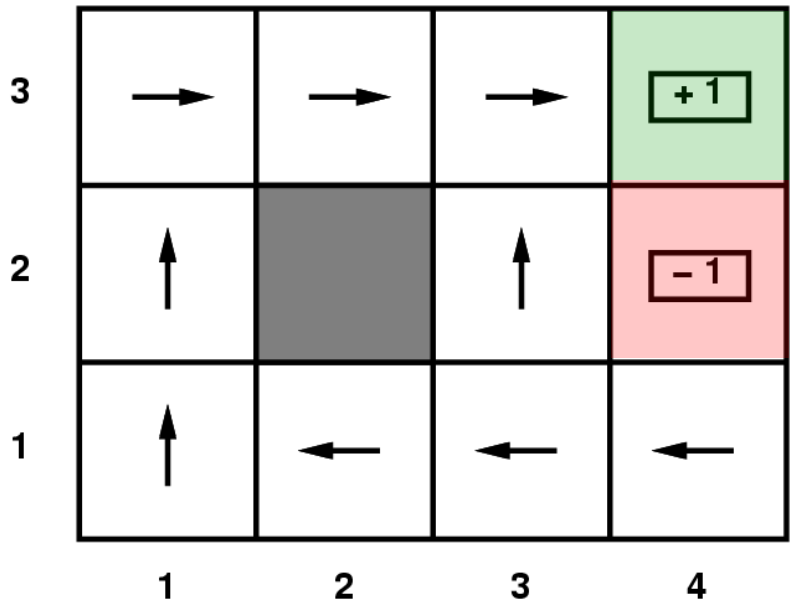
- Suppose given a stationary policy (shown by arrows)
  - Actions can stochastically lead to unintended grid cell
- Want to determine how good it is



Objective: Value Function

3	0.812	0.868	0.918	+1
2	0.762	▨	0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

Passive RL



- Estimate $V^{\pi}(s)$
- Not given
  - transition matrix, nor
  - reward function!
- Follow the policy for many epochs giving training sequences.

$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,4)$ <u>+1</u>
$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (3,4)$ <u>+1</u>
$(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2)$ <u>-1</u>

- Assume that after entering +1 or -1 state the agent enters zero reward terminal state
  - So we don't bother showing those transitions

Approach 1: Direct Estimation

- Direct estimation (also called Monte Carlo)
  - Estimate $V^{\pi}(s)$ as average total reward of epochs containing s (calculating from s to end of epoch)
- ***Reward to go*** of a state s

  the sum of the (discounted) rewards from that state until a terminal state is reached
- Key: use observed ***reward to go*** of the state as the direct evidence of the actual expected utility of that state
- Averaging the reward-to-go samples will converge to true value at state

Direct Estimation

- Converge very slowly to correct utilities values (requires a lot of sequences)

- Doesn't exploit Bellman constraints on policy values

$$V^{\pi}(s) = R(s) + \beta \sum_{s'} T(s,a,s') V^{\pi}(s')$$

  - It is happy to consider value function estimates that violate this property badly.

How can we incorporate the Bellman constraints?

Approach 2: Adaptive Dynamic Programming (ADP)

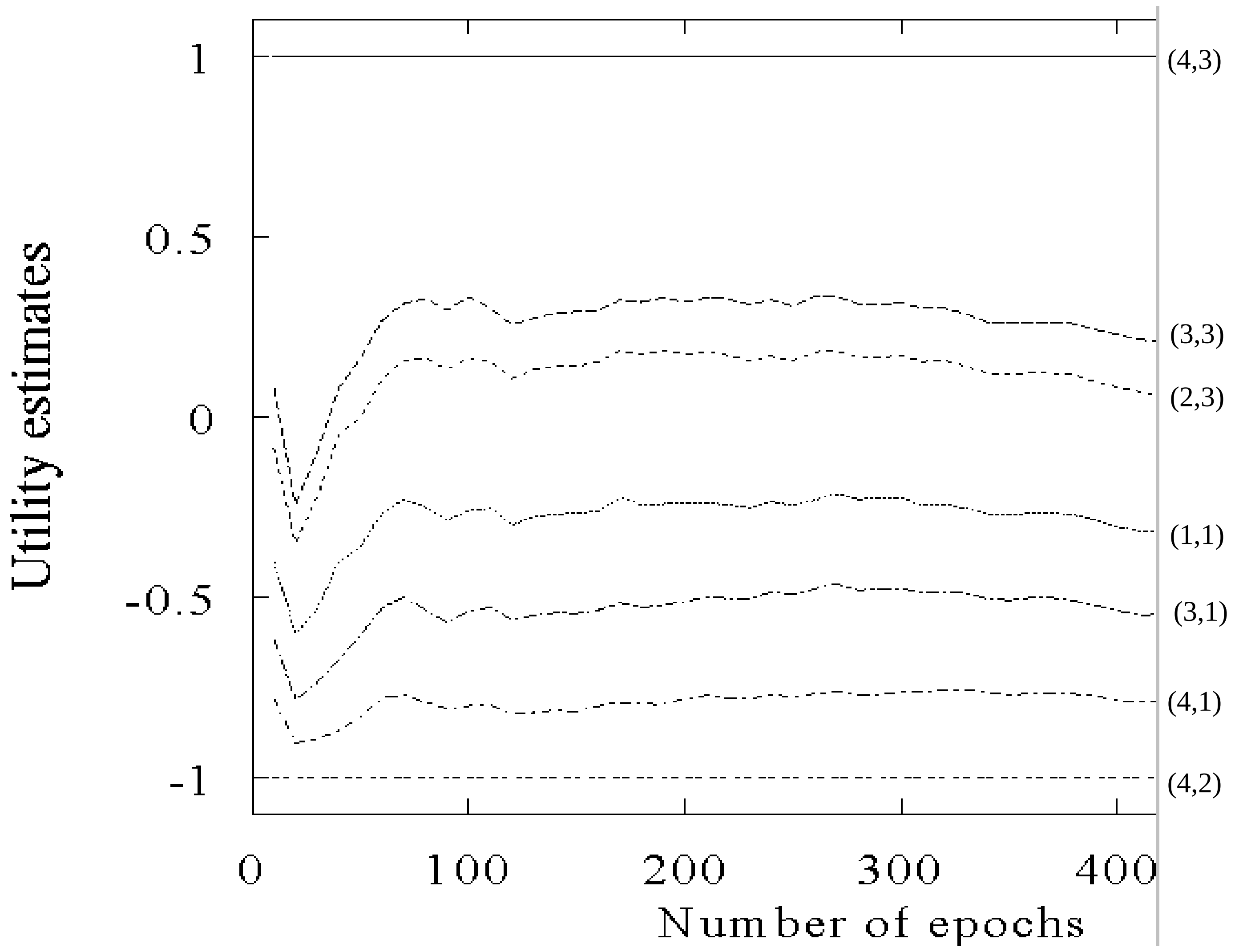
- ADP is a model based approach
  - Follow the policy for awhile
  - Estimate transition model based on observations
  - Learn reward function
  - Use estimated model to compute utility of policy

$$V^{\pi}(s) = \underset{\uparrow}{R(s)} + \beta \sum_{s'} \underset{\uparrow}{T(s,a,s')} V^{\pi}(s')$$

learned

- How can we estimate transition model T(s,a,s')?
  - Simply the fraction of times we see s' after taking a in state s.
  - NOTE: Can bound error with Chernoff bounds if we want

ADP learning curves



Approach 3: Temporal Difference Learning (TD)

- Can we avoid the computational expense of full DP policy evaluation?
- Temporal Difference Learning (model free)
  - Do local updates of utility/value function on a per-action basis
  - Don't try to estimate entire transition function!
  - For each transition from s to s', we perform the following update:

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\left(R(s) + \beta V^{\pi}(s') - V^{\pi}(s)\right)$$

updated estimate learning rate discount factor

- Intuitively moves us closer to satisfying Bellman constraint

$$V^{\pi}(s) = R(s) + \beta \sum_{s'} T(s,a,s') V^{\pi}(s')$$

Why?

Aside: Online Mean Estimation

- Suppose that we want to incrementally compute the mean of a sequence of numbers ($x_1, x_2, x_3, \ldots$)
  - E.g. to estimate the expected value of a random variable from a sequence of samples.

$$\hat{X}_{n+1} = \frac{1}{n+1} \sum_{i=1}^{n+1} x_i$$

average of n+1 samples

- Given a new sample x_{n+1}, the new mean is the old estimate (for n samples) plus the weighted difference between the new sample and old estimate

Aside: Online Mean Estimation

- Suppose that we want to incrementally compute the mean of a sequence of numbers ($x_{1,}$ $x_{2,}$ $x_{3,}$ ....)
  - E.g. to estimate the expected value of a random variable from a sequence of samples.

$$\hat{X}_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n+1}\left(x_{n+1} - \frac{1}{n}\sum_{i=1}^{n} x_i \right)$$

average of n+1 samples

Aside: Online Mean Estimation

- Suppose that we want to incrementally compute the mean of a sequence of numbers ($x_{1,}$ $x_{2,}$ $x_{3,}$ ….)
  - E.g. to estimate the expected value of a random variable from a sequence of samples.

$$\hat{X}_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n+1}\left(x_{n+1} - \frac{1}{n}\sum_{i=1}^{n} x_i \right)$$

$$= \hat{X}_n + \frac{1}{n+1}\left(x_{n+1} - \hat{X}_n \right)$$

average of n+1 samples

learning rate

sample n+1

- Given a new sample x_{n+1}, the new mean is the old estimate (for n samples) plus the weighted difference between the new sample and old estimate

Approach 3: Temporal Difference Learning (TD)

- TD update for transition from s to s':

$$\underset{\text{updated estimate}}{V^{\pi}(s)} \leftarrow V^{\pi}(s) + \underset{\text{learning rate}}{\alpha}\Big(\underbrace{R(s) + \beta V^{\pi}(s')}_{\text{(noisy) sample of value at s based on next state s'}} - V^{\pi}(s)\Big)$$

- So the update is maintaining a "mean" of the (noisy) value samples

- If the learning rate decreases appropriately with the number of samples (e.g. 1/n) then the value estimates will converge to true values! (non-trivial)

$$V^{\pi}(s) = R(s) + \beta \sum_{s'} T(s,a,s')V^{\pi}(s')$$

Approach 3: Temporal Difference Learning (TD)

- TD update for transition from s to s':

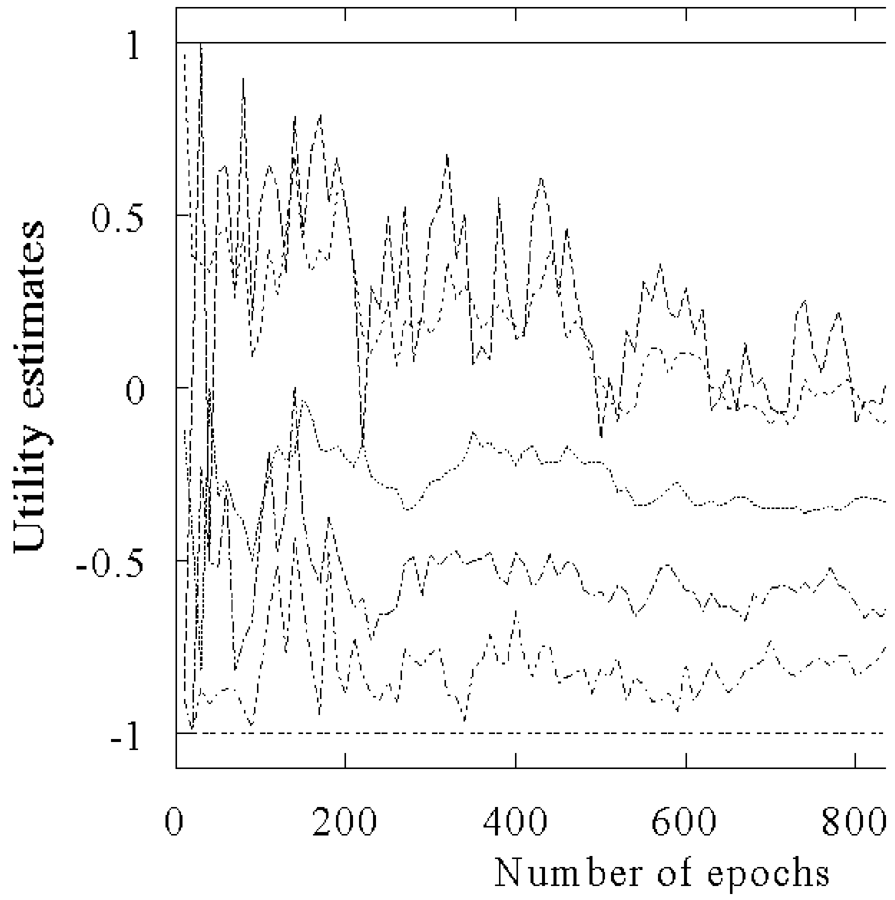
$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \overset{\text{learning rate}}{\alpha}(\underbrace{R(s) + \beta V^{\pi}(s')}_{\text{(noisy) sample of utility based on next state}} - V^{\pi}(s))$$

- Intuition about convergence
  - When V satisfies Bellman constraints then **expected** update is 0.

$$V^{\pi}(s) = R(s) + \beta \sum_{s'} T(s,a,s') V^{\pi}(s')$$

  - Can use results from stochastic optimization theory to prove convergence in the limit

The TD learning curve



- **Tradeoff:** requires more training experience (epochs) than ADP but much less computation per epoch
- Choice depends on relative cost of experience vs. computation

Passive RL: Comparisons

- Monte-Carlo Direct Estimation (model free)
  - Simple to implement
  - Each update is fast
  - Does not exploit Bellman constraints
  - Converges slowly
- Adaptive Dynamic Programming (model based)
  - Harder to implement
  - Each update is a full policy evaluation (expensive)
  - Fully exploits Bellman constraints
  - Fast convergence (in terms of updates)
- Temporal Difference Learning (model free)
  - Update speed and implementation similiar to direct estimation
  - Partially exploits Bellman constraints---adjusts state to ‘agree’ with observed successor
    - Not ***all*** possible successors as in ADP
  - Convergence in between direct estimation and ADP

Between ADP and TD

- Moving TD toward ADP
  - At each step perform TD updates based on observed transition and “imagined” transitions
  - Imagined transition are generated using estimated model

- The more imagined transitions used, the more like ADP
  - Making estimate more consistent with next state distribution
  - Converges in the limit of infinite imagined transitions to ADP

- Trade-off computational and experience efficiency
  - More imagined transitions require more time per step, but fewer steps of actual experience

Active Reinforcement Learning

- So far, we've assumed agent ***has*** a policy
  - We just learned how good it is
- Now, suppose agent must learn a good policy (ideally optimal)
  - While acting in uncertain world

Naïve Model-Based Approach

1. Act Randomly for a (long) time
   - Or systematically explore all possible actions
2. Learn
   - Transition function
   - Reward function
3. Use value iteration, policy iteration, …
4. Follow resulting policy thereafter.

Will this work? Yes (if we do step 1 long enough and there are no “dead-ends”)

Any problems? We will act randomly for a long time before exploiting what we know.

Revision of Naïve Approach

1. Start with initial (uninformed) model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by policy in current state
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the greedy policy suggested by current value estimate

Will this work? No. Can get stuck in local minima.
What can be done?

Exploration versus Exploitation

- Two reasons to take an action in RL
  - **Exploitation**: To try to get reward. We exploit our current knowledge to get a payoff.
  - **Exploration**: Get more information about the world. How do we know if there is not a pot of gold around the corner.
- To explore we typically need to take actions that do not seem best according to our current model.
- Managing the trade-off between exploration and exploitation is a critical issue in RL
- Basic intuition behind most approaches:
  - Explore more when knowledge is weak
  - Exploit more as we gain knowledge

ADP-based (model-based) RL

1. Start with initial model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Take action according to an explore/exploit policy (explores more early on and gradually uses policy from 2)
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the explore/exploit policy

Will this work? Depends on the explore/exploit policy. Any ideas?

Explore/Exploit Policies

- Greedy action is action maximizing estimated Q-value

$$Q(s,a) = R(s) + \beta \sum_{s'} T(s,a,s')V(s')$$

  - where V is current optimal value function estimate (based on current model), and R, T are current estimates of model
  - Q(s,a) is the expected value of taking action a in state s and then getting the estimated value V(s') of the next state s'

- Want an exploration policy that is greedy in the limit of infinite exploration (GLIE)
  - Guarantees convergence

- GLIE Policy 1
  - On time step t select random action with probability p(t) and greedy action with probability 1-p(t)
  - p(t) = 1/t will lead to convergence, but is slow

Explore/Exploit Policies

- GLIE Policy 1
  - On time step t select random action with probability p(t) and greedy action with probability 1-p(t)
  - p(t) = 1/t will lead to convergence, but is slow

- In practice it is common to simply set p(t) to a small constant ε (e.g. ε=0.1 or ε=0.01)
  - Called ε-greedy exploration

Explore/Exploit Policies

- GLIE Policy 2: Boltzmann Exploration
  - Select action a with probability,

$$\Pr(a \mid s) = \frac{\exp\left(Q(s,a)/T\right)}{\sum_{a' \in A} \exp\left(Q(s,a')/T\right)}$$

  - T is the temperature. Large T means that each action has about the same probability. Small T leads to more greedy behavior.
  - Typically start with large T and decrease with time

The Impact of Temperature

$$\Pr(a \mid s) = \frac{\exp\left(Q(s,a)/T\right)}{\sum_{a' \in A} \exp\left(Q(s,a')/T\right)}$$

- Suppose we have two actions and that Q(s,a1) = 1, Q(s,a2) = 2

- T=10 gives Pr(a1 | s) = 0.48, Pr(a2 | s) = 0.52
  - Almost equal probability, so will explore

- T= 1 gives Pr(a1 | s) = 0.27, Pr(a2 | s) = 0.73
  - Probabilities more skewed, so explore a1 less

- T = 0.25 gives Pr(a1 | s) = 0.02, Pr(a2 | s) = 0.98
  - Almost always exploit a2

Alternative Model-Based Approach: Optimistic Exploration

1. Start with initial model
2. Solve for “optimistic policy”
   (uses optimistic variant of value iteration)
   (inflates value of actions leading to unexplored regions)
3. Take greedy action according to optimistic policy
4. Update estimated model
5. Goto 2

Basically act as if all “unexplored” state-action pairs are maximally rewarding.

Optimistic Exploration

- Recall that value iteration iteratively performs the following update at all states:

$$V(s) \leftarrow R(s) + \beta \max_a \sum_{s'} T(s,a,s')V(s')$$

  - Optimistic variant adjusts update to make actions that lead to unexplored regions look good

- **Optimistic VI:** assigns highest possible value V^{max} to any state-action pair that has not been explored enough
  - Maximum value is when we get maximum reward forever

$$V^{\max} = \sum_{t=0}^{\infty} \beta^t R^{\max} = \frac{R^{\max}}{1-\beta}$$

- What do we mean by “explored enough”?
  - N(s,a) > N_e, where N(s,a) is number of times action a has been tried in state s and N_e is a user selected parameter

Optimistic Value Iteration

$$V(s) \leftarrow R(s) + \beta \max_a \sum_{s'} T(s,a,s')V(s')$$ Standard VI

- Optimistic value iteration computes an optimistic value function V$^+$ using following updates

$$V^+(s) \leftarrow R(s) + \beta \max_a \begin{cases} V^{\max}, & N(s,a) < N_e \\ \sum_{s'} T(s,a,s')V^+(s'), & N(s,a) \geq N_e \end{cases}$$

- The agent will behave initially as if there were wonderful rewards scattered all over around– **optimistic** .
- But after actions are tried enough times we will perform standard “non-optimistic” value iteration

Optimistic Exploration: Review

1. Start with initial model
2. Solve for optimistic policy using optimistic value iteration
3. Take greedy action according to optimistic policy
4. Update estimated model; Goto 2

Can any guarantees be made for the algorithm?

- If N_e is large enough and all state-action pairs are explored that many times, then the model will be accurate and lead to close to optimal policy
- But, perhaps some state-action pairs will never be explored enough or it will take a very long time to do so
- Optimistic exploration is equivalent to another algorithm, Rmax, which has been proven to efficiently converge

Another View of Optimistic Exploration: The Rmax Algorithm

1. Start with an optimistic model
   (assign largest possible reward to “unexplored states”)
   (actions from “unexplored states” only self transition)
2. Solve for optimal policy in optimistic model (standard VI)
3. Take greedy action according to policy
4. Update optimistic estimated model
   (if a state becomes “known” then use its true statistics)
5. Goto 2

Agent always acts greedily according to a model that assumes all “unexplored” states are maximally rewarding

Rmax: Optimistic Model

- Keep track of number of times a state-action pair is tried

- If N(s,a) < N_e then T(s,a,s)=1 and R(s) = Rmax in optimistic model,
- Otherwise T(s,a,s') and R(s) are based on estimates obtained from the N_e experiences (the estimate of true model)
  - For large enough N_e these will be accurate estimates

- An optimal policy for this optimistic model will try to reach unexplored states (those with unexplored actions) since it can stay at those states and accumulate maximum reward

- Never explicitly explores. Is always greedy, but with respect to an optimistic outlook.

Optimistic Exploration

- Rmax is equivalent to optimistic exploration via optimistic VI
  - Convince yourself of this.

- Is Rmax provably efficient?
  - If the model is every completely learned (i.e. N(s,a) > N_e, for all (s,a), then the policy will be near optimal
  - Recent results show that this will happen “quickly”

- **PAC Guarantee (Roughly speaking):** *There is a value of N_e (depending on n,m, and Rmax), such that with high probability the Rmax algorithm will select at most a polynomial number of action with value less than ε of optimal)*

- RL can be solved in poly-time in n, m, and Rmax!

TD-based Active RL

1. Start with initial value function
2. Take action from explore/exploit policy giving new state s' (should converge to greedy policy, i.e. GLIE)
3. Update estimated model
4. Perform TD update

$$V(s) \leftarrow V(s) + \alpha(R(s) + \beta V(s') - V(s))$$

   V(s) is new estimate of optimal value function at state s.

5. Goto 2

Just like TD for passive RL, but we follow explore/exploit policy

> Given the usual assumptions about learning rate and GLIE, TD will converge to an optimal value function!

TD-based Active RL

1. Start with initial value function
2. Take action from explore/exploit policy giving new state s’ (should converge to greedy policy, i.e. GLIE)
3. Update estimated model
4. Perform TD update

$$V(s) \leftarrow V(s) + \alpha(R(s) + \beta V(s') - V(s))$$

   V(s) is new estimate of optimal value function at state s.

5. Goto 2

Requires an estimated model. Why?

To compute the explore/exploit policy.

TD-Based Active Learning

- Explore/Exploit policy requires computing Q(s,a) for the exploit part of the policy
  - Computing Q(s,a) requires T and R in addition to V
- Thus TD-learning must still maintain an estimated model for action selection
- It is computationally more efficient at each step compared to Rmax (i.e. optimistic exploration)
  - TD-update vs. Value Iteration
  - But model requires much more memory than value function
- Can we get a model-fee variant?

Q-Learning: Model-Free RL

- Instead of learning the optimal value function V, directly learn the optimal Q function.
  - Recall Q(s,a) is the expected value of taking action a in state s and then following the optimal policy thereafter
- Given the Q function we can act optimally by selecting action greedily according to Q(s,a) without a model
- The optimal Q-function satisfies $V(s) = \max_{a'} Q(s,a')$ which gives:

$$Q(s,a) = R(s) + \beta \sum_{s'} T(s,a,s') V(s')$$

$$= R(s) + \beta \sum_{s'} T(s,a,s') \max_{a'} Q(s',a')$$

How can we learn the Q-function directly?

Q-Learning: Model-Free RL

Bellman constraints on optimal Q-function:

$$Q(s,a) = R(s) + \beta \sum_{s'} T(s,a,s') \max_{a'} Q(s,a')$$

- We can perform updates after each action just like in TD.
  - After taking action a in state s and reaching state s' do: (note that we directly observe reward R(s))

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\underbrace{R(s) + \beta \max_{a'} Q(s',a')}_{\text{(noisy) sample of Q-value based on next state}} - Q(s,a))$$

Q-Learning

1. Start with initial Q-function (e.g. all zeros)
2. Take action from explore/exploit policy giving new state s' (should converge to greedy policy, i.e. GLIE)
3. Perform TD update

   $$Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \beta \max_{a'} Q(s',a') - Q(s,a)\right)$$

   Q(s,a) is current estimate of optimal Q-function.
4. Goto 2

- Does not require model since we learn Q directly!
- Uses explicit |S|x|A| table to represent Q
- Explore/exploit policy directly uses Q-values
  - E.g. use Boltzmann exploration.
  - Book uses exploration function for exploration (Figure 21.8)

Q-Learning: Speedup for Goal-Based Problems

- **Goal-Based Problem:** receive big reward in goal state and then transition to terminal state
  - Mini-project 2 is goal based
- Consider initializing Q(s,a) to zeros and then observing the following sequence of (state, reward, action) triples
  - (s0,0,a0) (s1,0,a1) (s2,10,a2) (terminal,0)
- The sequence of Q-value updates would result in: Q(s0,a0) = 0, Q(s1,a1) =0, Q(s2,a2)=10
- So nothing was learned at s0 and s1
  - Next time this trajectory is observed we will get non-zero for Q(s1,a1) but still Q(s0,a0)=0

Q-Learning: Speedup for Goal-Based Problems

- From the example we see that it can take many learning trials for the final reward to "back propagate" to early state-action pairs

- Two approaches for addressing this problem:
  1. **Trajectory replay**: store each trajectory and do several iterations of Q-updates on each one
  2. **Reverse updates:** store trajectory and do Q-updates in reverse order

- In our example (with learning rate and discount factor equal to 1 for ease of illustration) reverse updates would give
  - Q(s2,a2) = 10, Q(s1,a1) = 10, Q(s0,a0)=10

Q-Learning: Suggestions for Mini Project 2

- A very simple exploration strategy is ε-greedy exploration (generally called "epsilon greedy")
  - Select a "small" value for e (perhaps 0.1)
  - On each step:
    - With probability ε select a random action, and with probability 1- ε select a greedy action
- But it might be interesting to play with exploration a bit (e.g. compare to a decreasing exploration rate)
- You can use a discount factor of one or close to 1.

Active Reinforcement Learning Summary

- Methods
  - ADP
  - Temporal Difference Learning
  - Q-learning
- All converge to optimal policy assuming a GLIE exploration strategy
  - Optimistic exploration with ADP can be shown to converge in polynomial time with high probability
- All methods assume the world is not too dangerous (no cliffs to fall off during exploration)
- So far we have assumed small state spaces

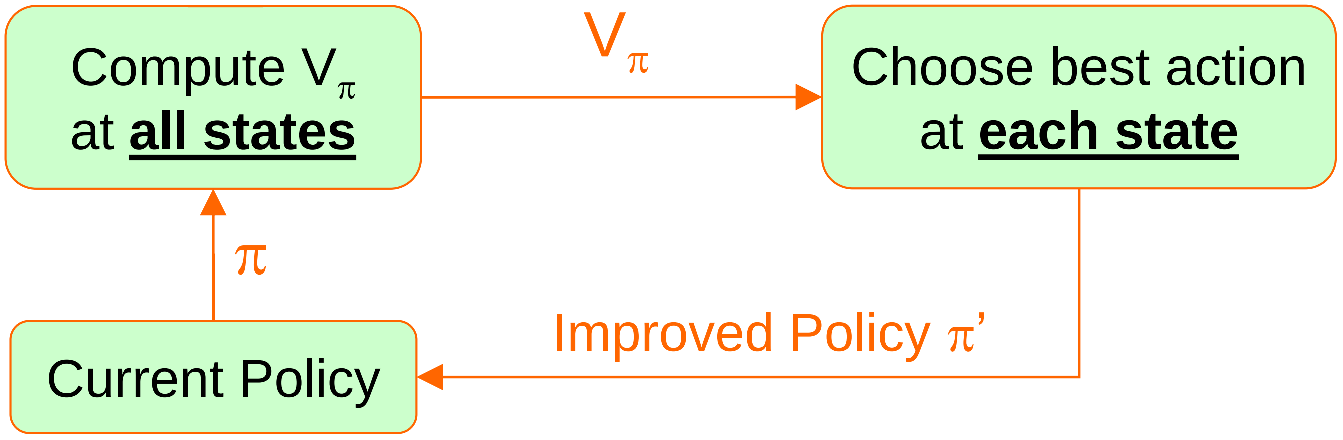
ADP vs. TD vs. Q

- Different opinions.
- (my opinion) When state space is small then this is not such an important issue.
- Computation Time
  - ADP-based methods use more computation time per step
- Memory Usage
  - ADP-based methods uses O(mn^2) memory
  - Active TD-learning uses O(mn^2) memory (must store model)
  - Q-learning uses O(mn) memory for Q-table
- Learning efficiency (performance per unit experience)
  - ADP-based methods make more efficient use of experience by storing a model that summarizes the history and then reasoning about the model (e.g. via value iteration or policy iteration)

What about large state spaces?

- One approach is to map the original state space S to a much smaller state space S' via some hashing function.
  - Ideally "similar" states in S are mapped to the same state in S'
- Then do learning over S' instead of S.
  - Note that the world may not look Markovian when viewed through the lens of S', so convergence results may not apply
  - But, still the approach can work if a good enough S' is engineered (requires careful design), e.g.
  - *Empirical Evaluation of a Reinforcement Learning Spoken Dialogue System.* With S. Singh, D. Litman, M. Walker. *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000
- We will now study three other approaches for dealing with large state-spaces
  - Value function approximation
  - Policy gradient methods
  - Least Squares Policy Iteration

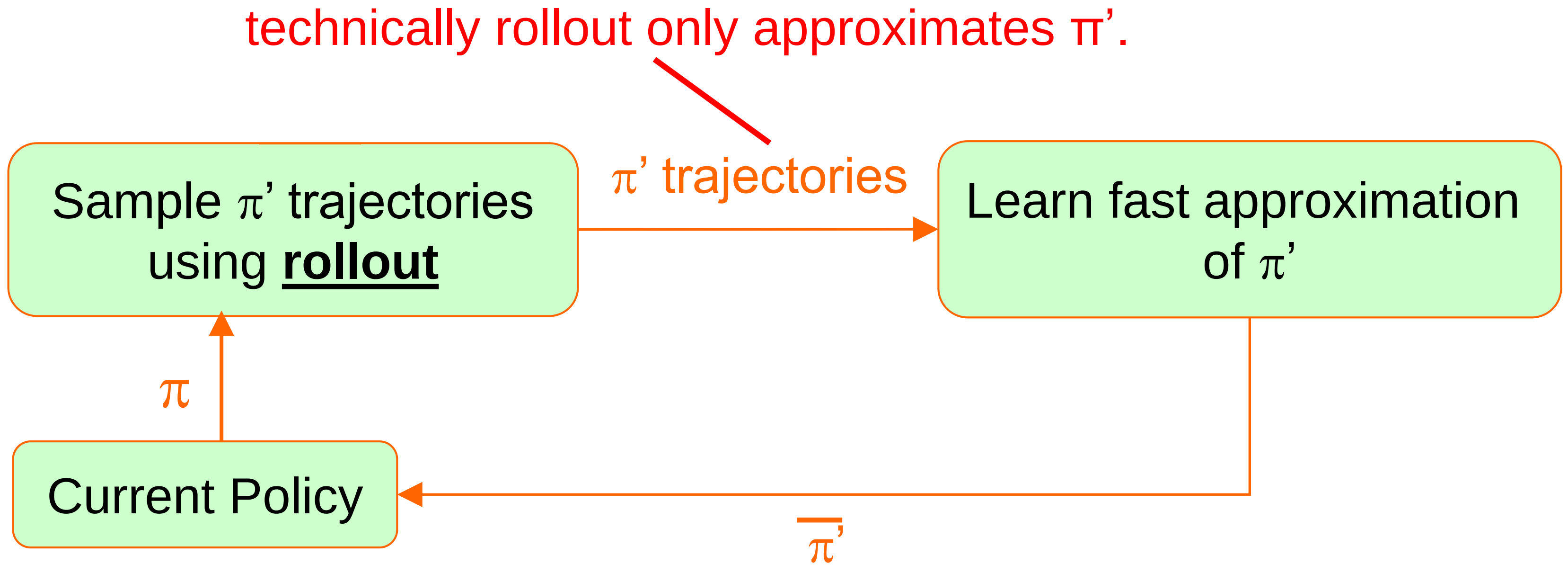
Return to Policy Iteration


Compute V_π at **all states**
V_π
Choose best action at **each state**
π
Improved Policy π'
Current Policy

Approximate policy iteration:

- Only computes values and improved action at some states.
- Uses those to infer a fast, compact policy over all states.

Approximate Policy Iteration

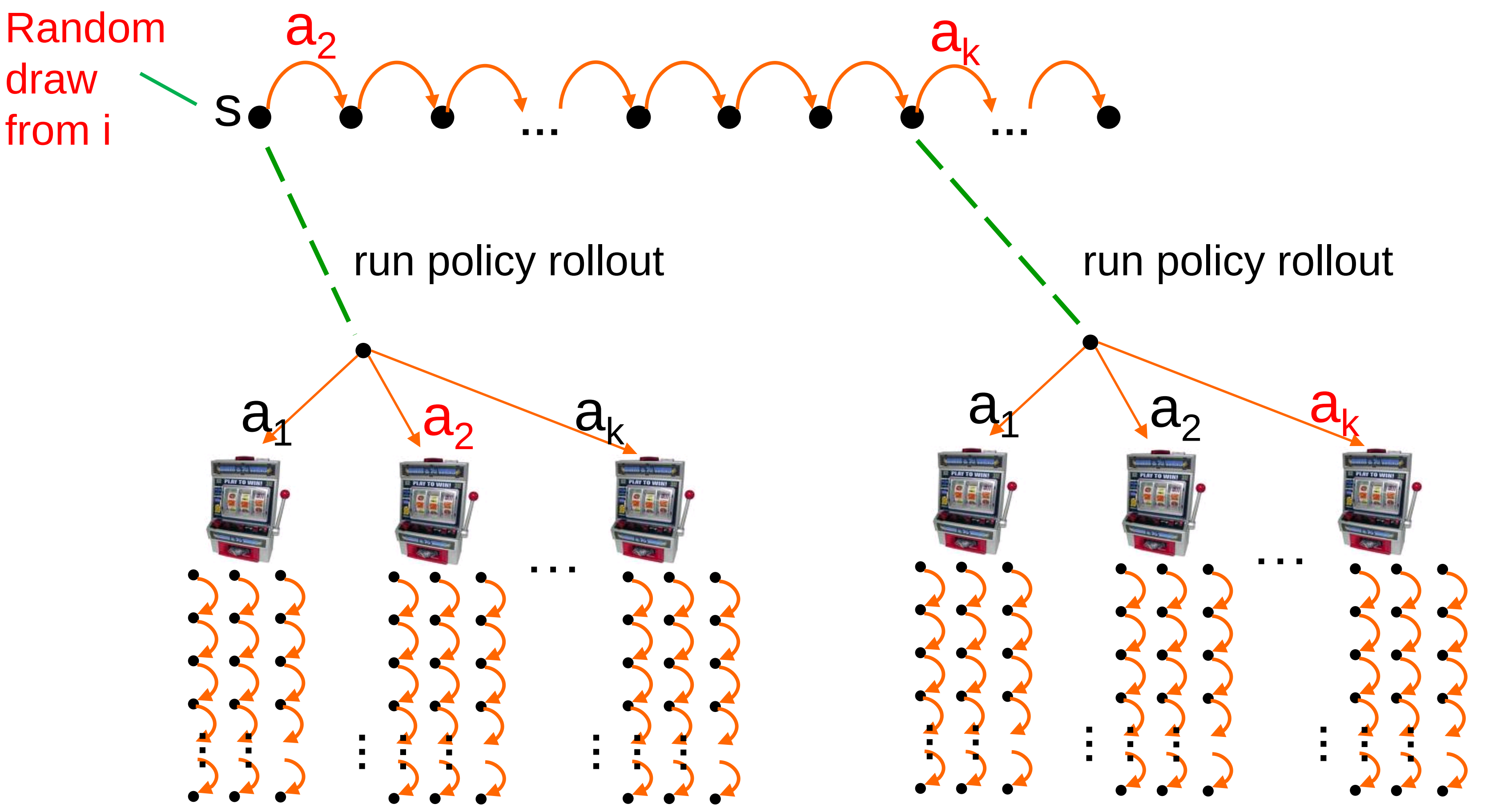

technically rollout only approximates π'.
Sample π' trajectories using **rollout**
π' trajectories
Learn fast approximation of π'
π
Current Policy
$\overline{\pi'}$

1. Generate trajectories of rollout policy
   (starting state of each trajectory is drawn from initial state distribution I)
2. "Learn a fast approximation" of rollout policy
3. Loop to step 1 using the learned policy as the base policy

What do we mean by generate trajectories?

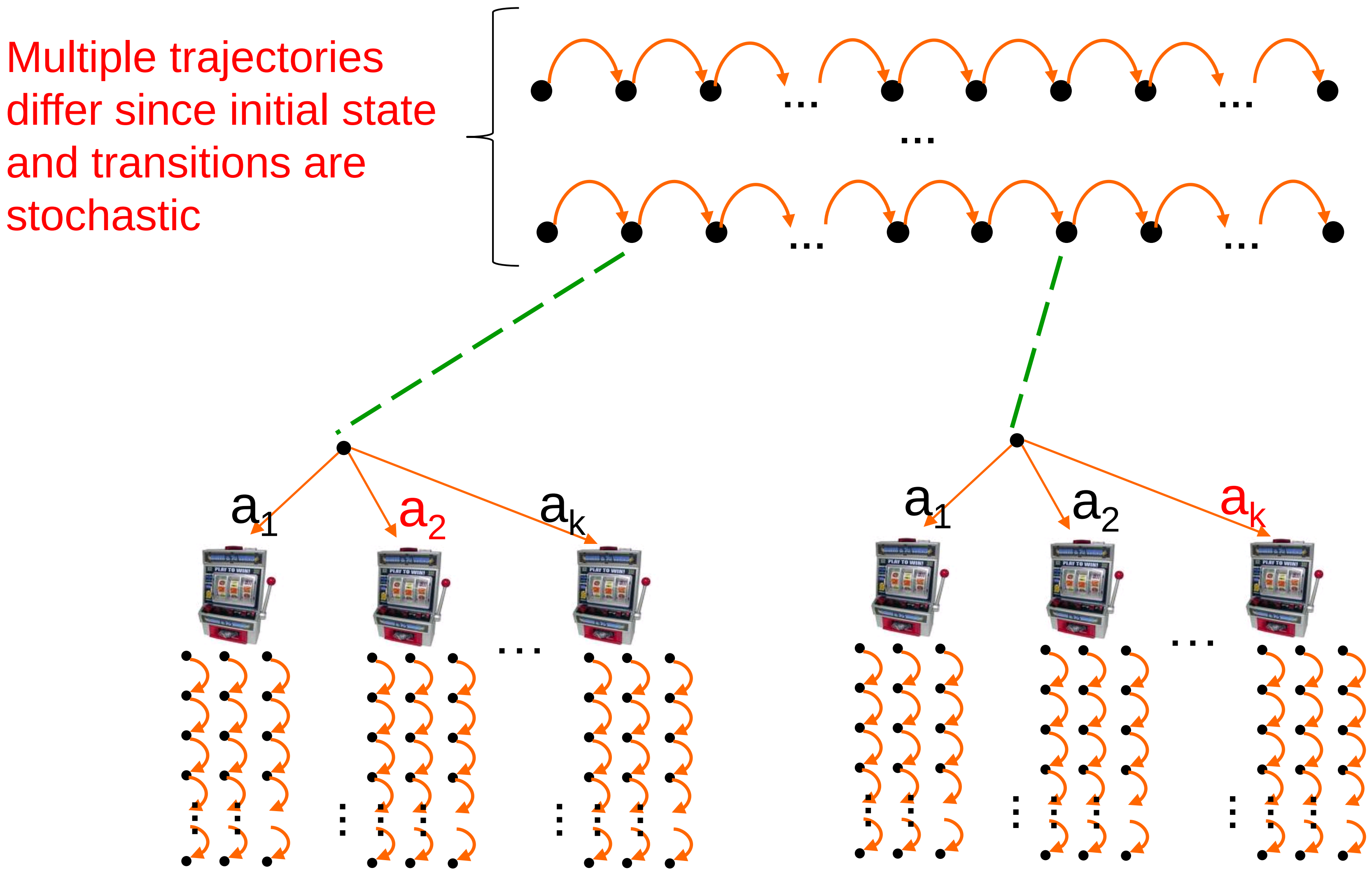
Generating Rollout Trajectories

Get trajectories of current rollout policy from an initial state



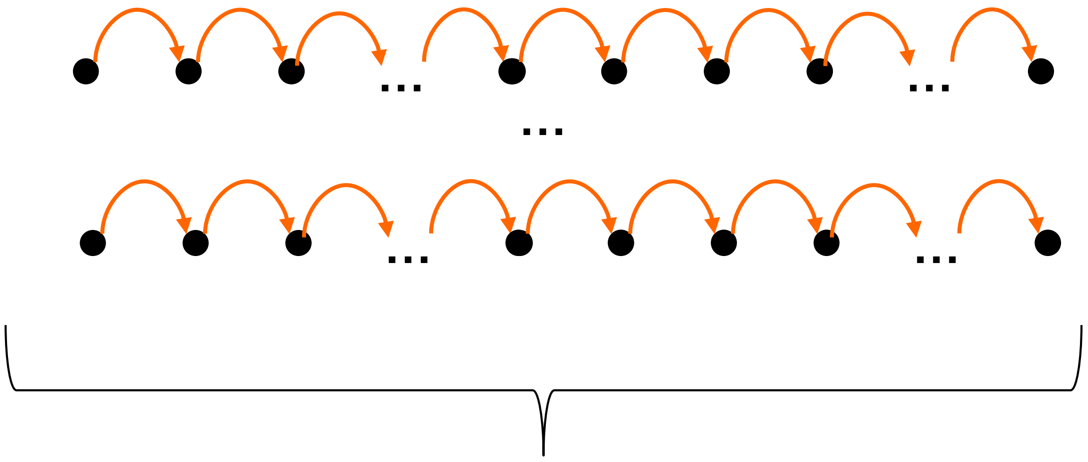
Generating Rollout Trajectories

Get trajectories of current rollout policy from an initial state


Multiple trajectories differ since initial state and transitions are stochastic
a_1
a_2
a_k
a_1
a_2
a_k

Generating Rollout Trajectories

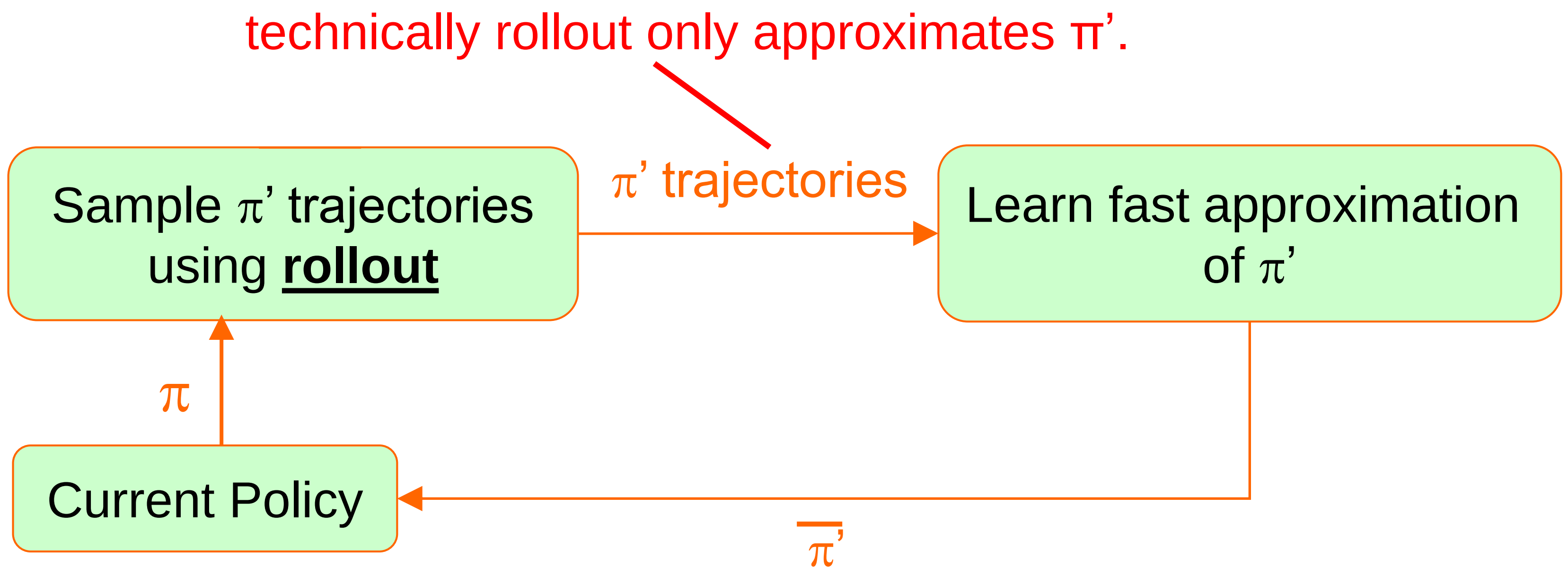
Get trajectories of current rollout policy from an initial state



Results in a set of state-action pairs giving the action selected by "improved policy" in states that it visits.

$$\{(s_1,a_1),\ (s_2,a_2),\ldots,(s_n,a_n)\}$$

Approximate Policy Iteration


technically rollout only approximates π'.
Sample π' trajectories using **rollout**
π' trajectories
Learn fast approximation of π'
π
Current Policy
$\overline{\pi}'$

1. Generate trajectories of rollout policy
   (starting state of each trajectory is drawn from initial state distribution I)
2. "Learn a fast approximation" of rollout policy
3. Loop to step 1 using the learned policy as the base policy

What do we mean by "learn an approximation"?

Aside: Classifier Learning

- A **classifier** is a function that labels inputs with class labels.
- “Learning” classifiers from training data is a well studied problem (decision trees, support vector machines, neural networks, etc).

$$\begin{tikzcd}
\text{Training Data} \\ \{(x_1,c_1),(x_2,c_2),\ldots,(x_n,c_n)\} \arrow[d] \\ \boxed{\text{Learning Algorithm}} \arrow[d] \\ \text{Classifier} \\ H : X \rightarrow C
\end{tikzcd}$$

Example problem:

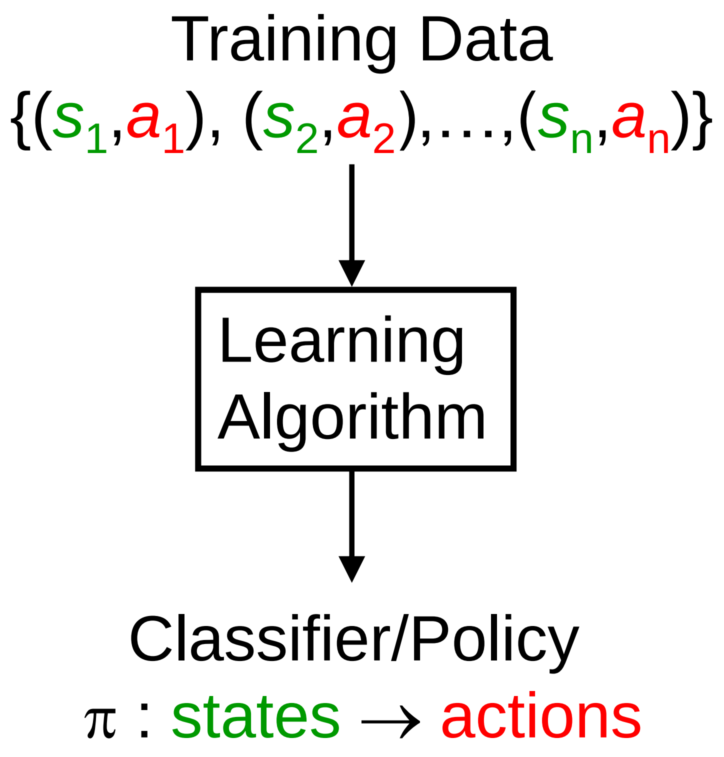
x_i - image of a face

$c_i \in$ {male, female}

Aside: Control Policies are Classifiers

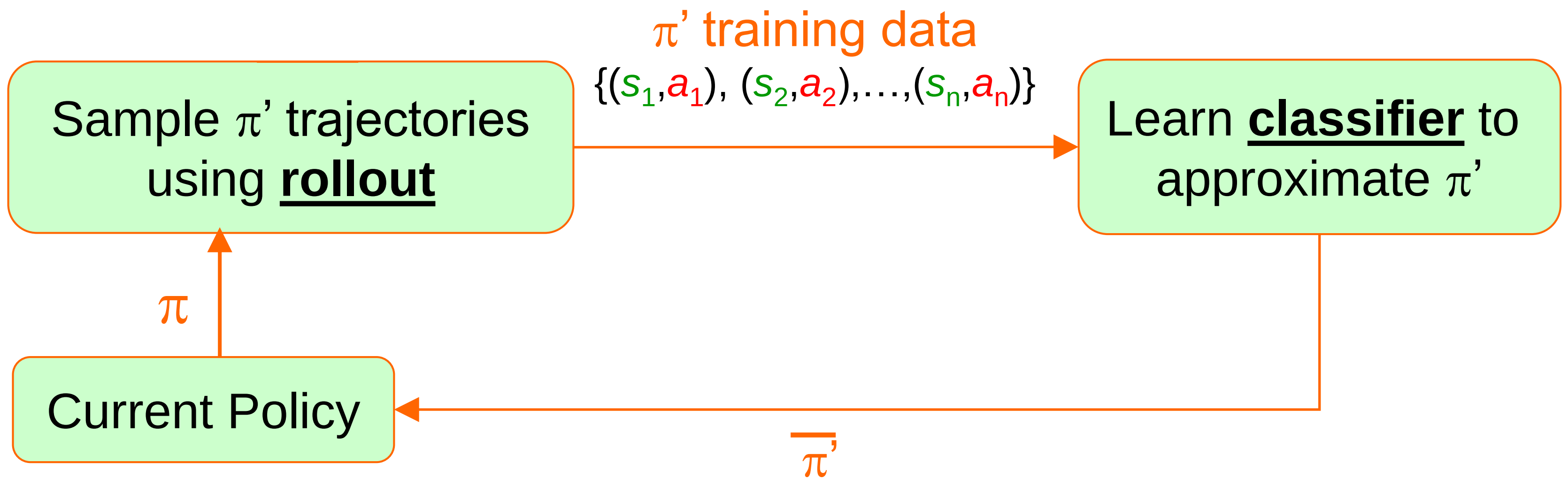
A **control policy** maps states and goals to actions.

π : states $\rightarrow$ actions



Training Data
$\{(s_1,a_1), (s_2,a_2),\ldots,(s_n,a_n)\}$
Learning Algorithm
Classifier/Policy
π : states $\rightarrow$ actions

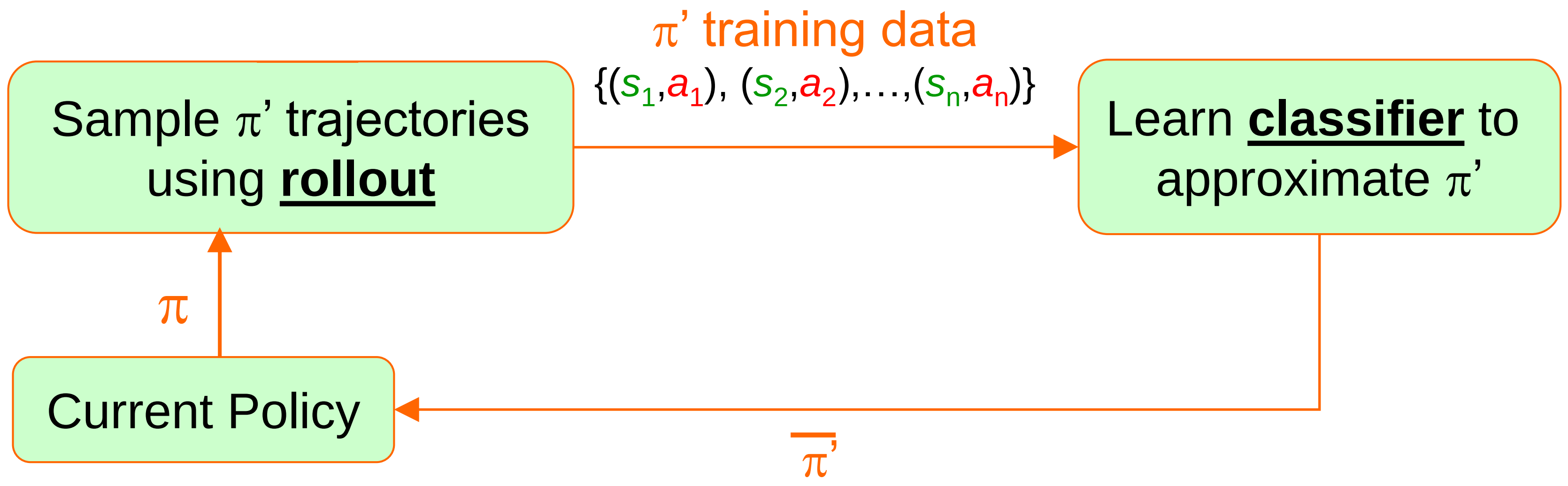
Approximate Policy Iteration


π' training data
{(s₁,a₁), (s₂,a₂),…,(sₙ,aₙ)}
Sample π' trajectories using **rollout**
Learn **classifier** to approximate π'
π
Current Policy
π̄'

1. Generate trajectories of rollout policy
   Results in training set of state-action pairs along trajectories

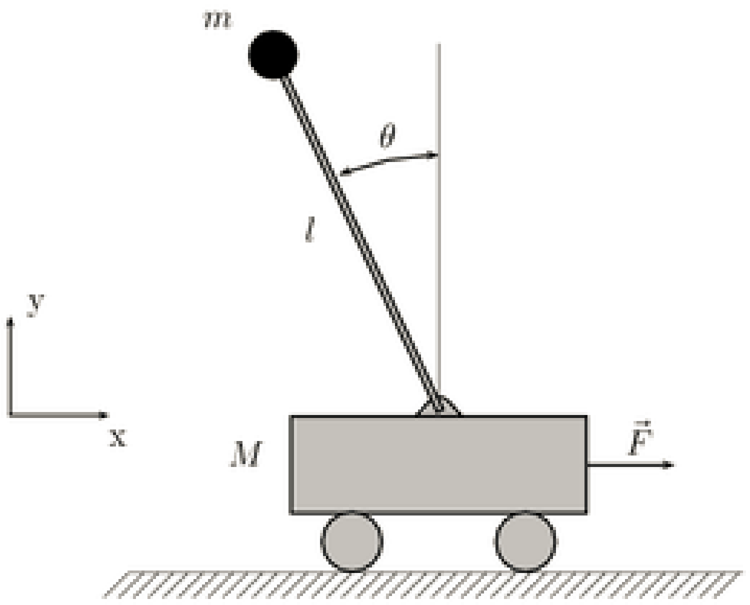
   $$T = \{(s_1,a_1), (s_2,a_2),\ldots,(s_n,a_n)\}$$
2. Learn a classifier based on *T* to approximate rollout policy
3. Loop to step 1 using the learned policy as the base policy

Approximate Policy Iteration


π' training data
{(s1,a1), (s2,a2),…,(sn,an)}
Sample π' trajectories using rollout
Learn classifier to approximate π'
π
Current Policy
π̄'

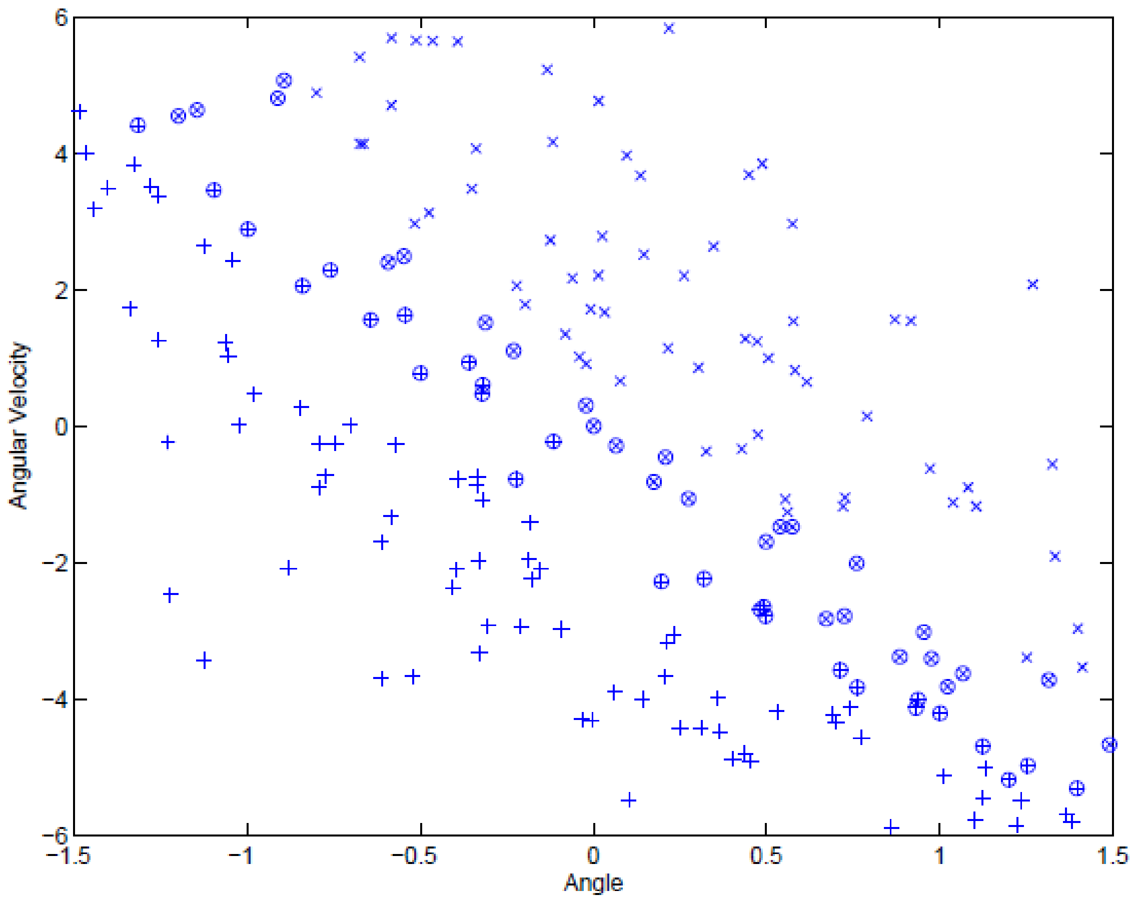
- The hope is that the learned classifier will capture the general structure of improved policy from examples
- Want classifier to quickly select correct actions in states outside of training data (classifier should generalize)
- Approach allows us to leverage large amounts of work in machine learning

API for Inverted Pendulum



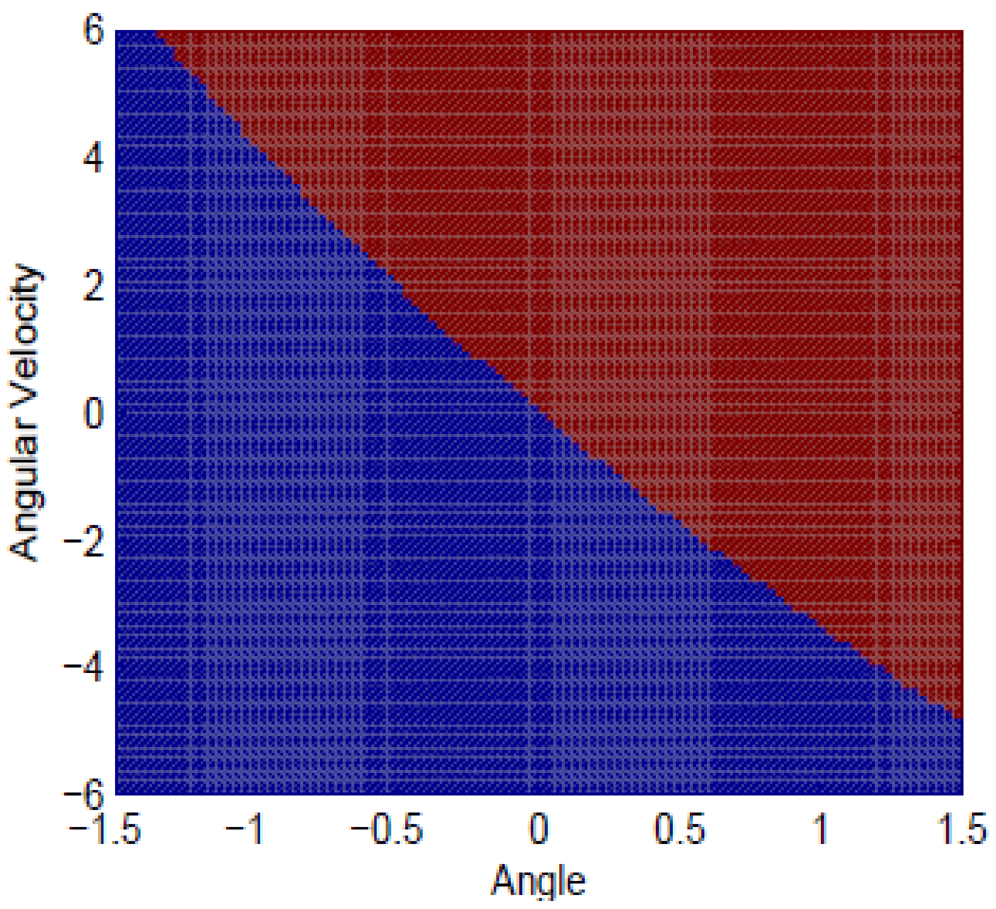
Consider the problem of balancing a pole by applying either a positive or negative force to the cart. The state space is described by the velocity of the cart and angle of the pendulum.
There is noise in the force that is applied, so problem is stochastic.

Experimental Results



A data set from an API iteration. + is positive action, x is negative (ignore the circles in the figure)

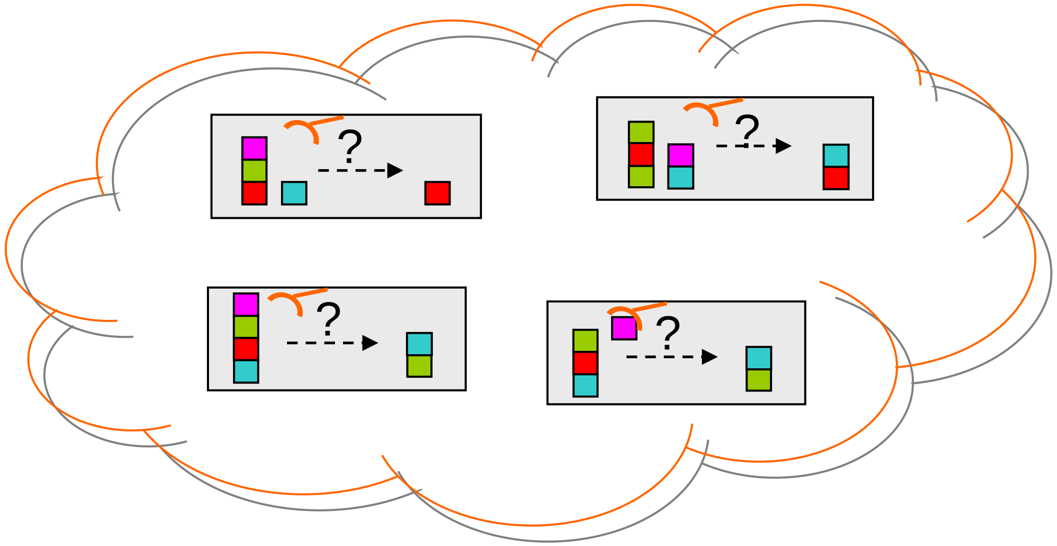
Experimental Results



Support vector machine used as classifier.
(take CS534 for details)
Maps any state to + or –

Learned classifier/policy after 2 iterations: (near optimal)
blue = positive, red = negative

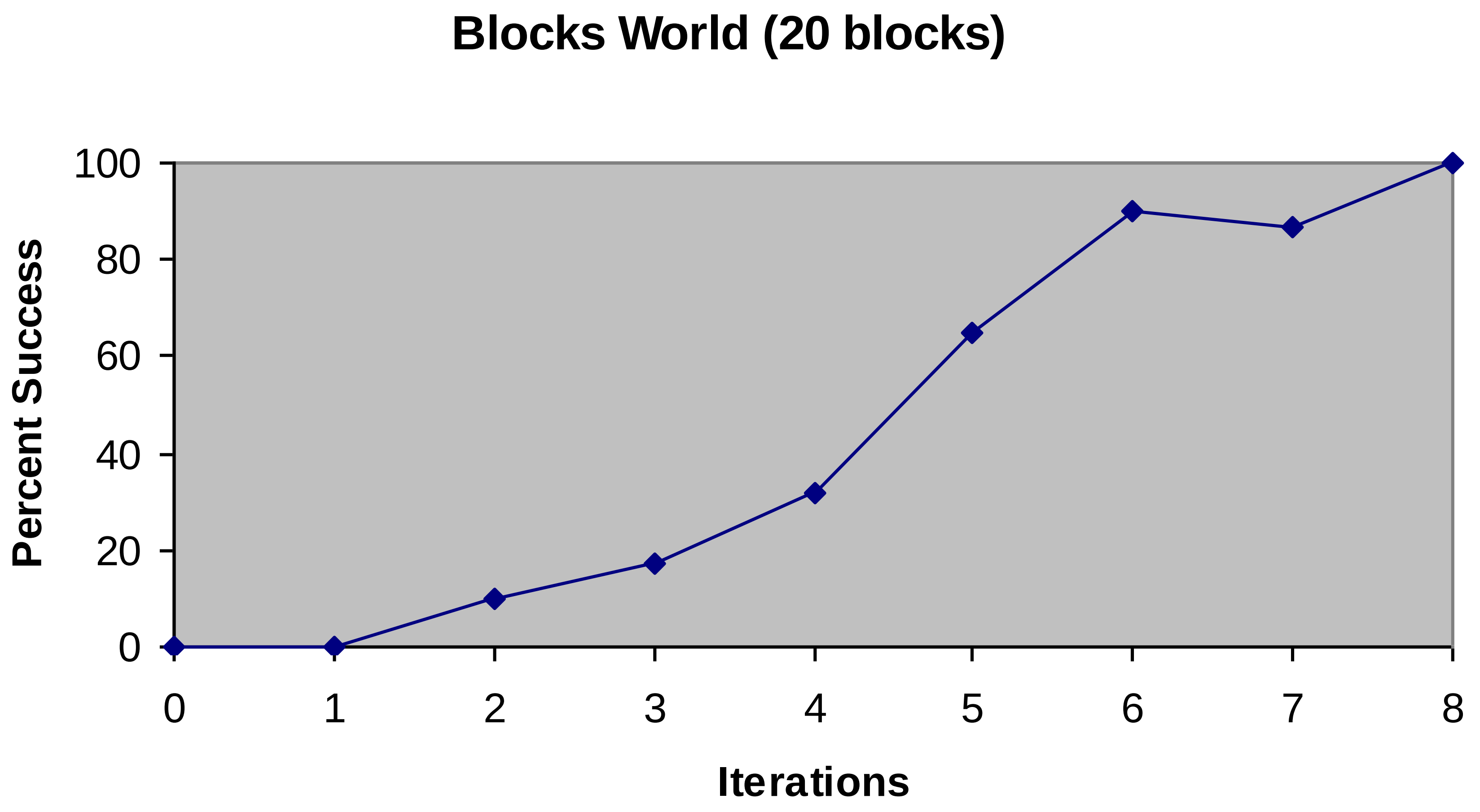
API for Stacking Blocks



Consider the problem of form a goal configuration of blocks/crates/etc. from a starting configuration using basic movements such as pickup, putdown, etc.

Also handle situations where actions fail and blocks fall.

Experimental Results



The resulting policy is fast near optimal. These problems are very hard for more traditional planners.

Summary of API

- Approximate policy iteration is a practical way to select policies in large state spaces
- Relies on ability to learn good, compact approximations of improved policies (must be efficient to execute)
- Relies on the effectiveness of rollout for the problem

- There are only a few positive theoretical results
  - convergence in the limit under strict assumptions
  - PAC results for single iteration
- But often works well in practice