

# Decision-Theoretic Planning and Learning in Relational Domains

Prasad Tadepalli and Alan Fern  
 Oregon State University  
 Kristian Kersting  
 Massachusetts Institute of Technology

• Thanks to Craig Boutilier, Kurt Driessens, Thomas Gartner, Roni Kharon, Changgang Wang, Daniel Weld



## Overview

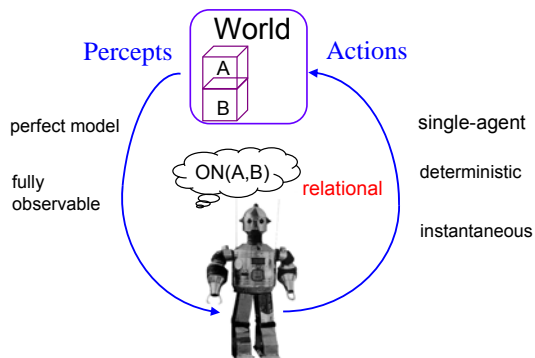
- Markov Decision Processes (MDPs)
  - ▲ Dynamic Programming
    - Value Iteration
    - Policy Iteration
  - ▲ Reinforcement Learning
    - TD-Learning
    - Q-Learning
- Relational Reinforcement Learning
  - ▲ Value Function Learning
    - Propositionalization
    - Relational Regression
  - ▲ Relational Policy Learning
    - Approximate Policy Iteration
- Planning in Relational MDPs
  - ▲ Symbolic Dynamic Programming
    - ReBel: Relational Bellman update operator
    - First Order Decision Diagrams
  - ▲ First-Order Approximate Linear Programming

# Markov Decision Processes

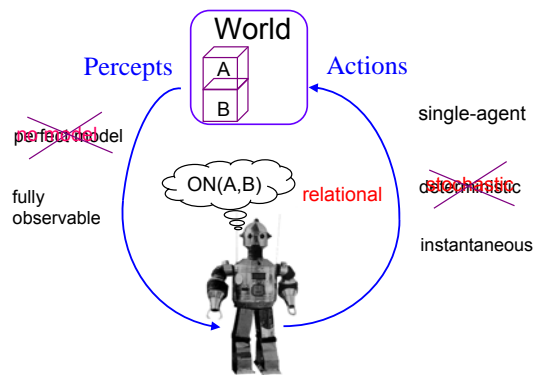
## Markov Decision Processes

- MDP Basics
  - ▲ States, actions, rewards
  - ▲ Policies and their values
- Dynamic Programming
  - ▲ Value Iteration
  - ▲ Policy Iteration
- Reinforcement Learning
  - ▲ TD-Learning
  - ▲ Q-Learning

## Classical Planning Assumptions



## Markov Decision Process (MDP) Model



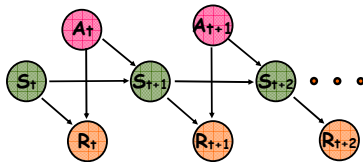
## Markov Decision Processes

- An MDP has four components: **S, A, R, T**:
  - ▶ (finite) state set  $S$  ( $|S| = n$ )
  - ▶ (finite) action set  $A$  ( $|A| = m$ )
  - ▶ (Markov) transition function  $T(s, a, s') = \Pr(s' | s, a)$ 
    - Probability of going to state  $s'$  after taking action  $a$  in state  $s$
  - ▶ bounded, real-valued reward function  $R(s, a)$ 
    - Immediate reward we get for executing action  $a$  in state  $s$
    - Can be generalized to be a stochastic function
  - ▶ Can generalize to countable or continuous state and action spaces (but algorithms will be different)

## Assumptions

- **Markovian dynamics** (history independence)
  - ▶ Next state only depends on current state and current action
- **Reward is a deterministic function of current state and action**
  - ▶ Generalizes to stochastic, but Markovian, reward
- **Stationary dynamics and reward**
  - ▶ The world dynamics do not depend on time
- **Full observability**
  - ▶ The agent knows exactly what state it is in

## Graphical View of MDP



## Policies (“plans” for MDPs)

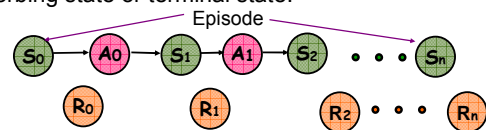
- Stationary policy
  - ▶  $\pi: S \rightarrow A$
  - ▶  $\pi(s)$  is action to do at state  $s$  (regardless of time)
  - ▶ specifies a continuously reactive controller
- These assume or have these properties:
  - ▶ full observability
  - ▶ history-independence
  - ▶ deterministic action choice

## Policies and Value Functions

- Stationary policy – specifies a reactive controller
  - ▶  $\pi: S \rightarrow A$
  - ▶  $\pi(s)$  is action to do at state  $s$  (regardless of time)
- **Value function**  $V: S \rightarrow \mathbb{R}$  associates values with states
- $V_\pi(s)$  denotes some measure of “accumulated reward” when starting from state  $s$  and following policy  $\pi$ 
  - ▶ Depends on immediate reward, but also what you achieve subsequently by following  $\pi$
  - ▶ 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 policy (method depends on how we define value)

## Episodic MDPs

- The MDP has an initial state distribution.
- Every policy eventually leads to a zero-cost absorbing state or terminal state.



- Value of a policy is the *expected* total reward during one episode of the policy:

$$V_\pi(s) = E(R_0 + \dots + R_n).$$



## Bellman Equation for Optimal Policies

- **Optimal policy:** Policy whose value function dominates all others:  $\forall s, \pi, V_{\pi^*}(s) \geq V_{\pi}(s)$
- Optimal policies exist and satisfy the Bellman equation (Howard 1960):

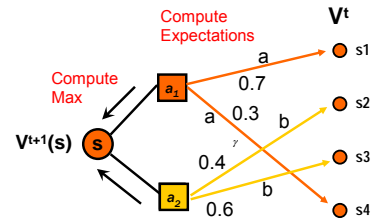
$$V_{\pi^*}(s) = 0 \text{ for all absorbing states}$$

$$V_{\pi^*}(s) = \text{Max}_a R(s, a) + \gamma \sum_{s'} T(s, a, s') \cdot V_{\pi^*}(s')$$

immediate reward      Best expected future payoff

## Bellman Backup

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



$$V^{t+1}(s) = \max [ R(s, a) + \gamma \{ 0.7 V^t(s1) + 0.3 V^t(s2) \} \\ R(s, b) + \gamma \{ 0.4 V^t(s2) + 0.6 V^t(s3) \} ]$$

## Value Iteration

- Markov property allows exploitation of DP principle for optimal policy construction
  - ↳ no need to enumerate  $|A|^T$  possible policies

- Value Iteration

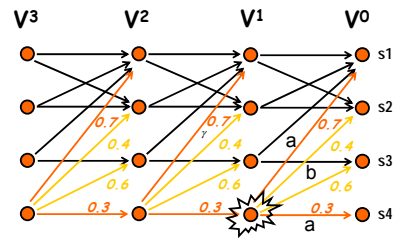
$$V^0(s) = 0, \quad \forall s$$

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

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

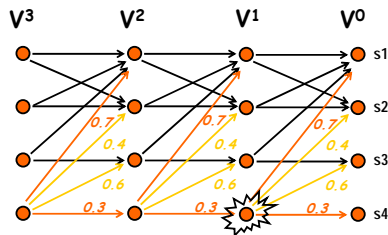
$V^t$  is an approximation to optimal value function  
 $\pi^*(s)$  is optimal policy

## Synchronous Value Iteration



$$V^1(s4) = \max \{ R(s4, a) + \gamma [ 0.7 V^0(s1) + 0.3 V^0(s4) ] \\ R(s4, b) + \gamma [ 0.4 V^0(s2) + 0.6 V^0(s3) ] \}$$

## Synchronous Value Iteration



$$\Pi^*(s4, t) = \text{argmax} \{ \blacksquare, \blacksquare \}$$

## Computing an Optimal Value Function

- **Bellman equation** for optimal value function
 
$$V^*(s) = R(s, a) + \gamma \max_a \sum_{s'} T(s, a, s') \cdot V^*(s')$$
- **Value Iteration** – synchronously update the values of all states: Apply “Bellman Operator” to the value function.

$$V^0(s) = 0$$

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

- Convergence guaranteed due to the contraction properties of the Bellman operator:  $\|V^{t+1} - V^*\| < \gamma \|V^t - V^*\|$

## Policy Iteration

- Given fixed policy, can compute its value exactly by solving simultaneous linear equations:

$$V_{\pi}(s) = R(s, a) + \gamma \sum_{s'} T(s, \pi(s), s') \cdot V_{\pi}(s')$$

- Alternate between policy evaluation and policy improvement

1. Choose a random policy  $\pi$

2. Loop:

(a) Evaluate  $V_{\pi}$

(b) For each  $s$  in  $S$ , set  $\pi'(s) = \arg \max_a \sum_{s'} T(s, a, s') \cdot V_{\pi}(s')$

(c) Replace  $\pi$  with  $\pi'$

Until no improving action possible at any state

Policy improvement

## Policy Iteration Notes

- Each step of policy iteration is guaranteed to strictly improve the policy at some state when improvement is possible
- Convergence assured (Howard)
  - intuitively: no local maxima in value space, and each policy must improve value; since finite number of policies, will converge to optimal policy
- Gives exact value of optimal policy

## Value Iteration vs. Policy Iteration

- Both are guaranteed to converge.
- Which is faster? VI or PI
  - It depends on the problem
- VI takes more iterations than PI, but PI requires more time on each iteration
  - PI must perform policy evaluation on each step which involves solving a linear system
- Complexity:
  - There are at most  $\exp(n)$  policies, so PI is no worse than exponential time in number of states
  - Empirically  $O(n)$  iterations are required
  - Still no polynomial bound on the number of iterations (open problem)!

## Linear Programming

- Minimize  $V(s_0)$  subject to:

$$\forall s, a, V(s) \geq R(s, a) + \gamma \sum_{s'} T(s, a, s') \cdot V(s')$$

- Note the relationship to the Bellman Equation

$$V^*(s) = \text{Max}_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') \cdot V^*(s')]$$

- Minimizing  $V(s_0)$  subject to the constraints ensures that there exists at least one (optimal) action for each state for which the constraints are tight, i.e., l.h.s. = r.h.s. satisfying the Bellman equation.

## Roadmap

- MDP Basics
  - States, actions, rewards
  - Policies and their values
- Dynamic Programming
  - Value Iteration
  - Policy Iteration
- Reinforcement Learning
  - TD-Learning
  - Q-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 **stochastic effects**
    - Which are initially unknown
  - Rewards / punishments** are infrequent
    - Often at the end of long sequences of actions
    - Credit/blame assignment: How do we determine what action(s) were really responsible for reward or punishment?
- Situated agent: learner **must decide** what actions to take at each step

## 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
  - ▲ can use value iteration or policy iteration treating the learned model as if it is correct
  - ▲ can do sample-based update of the value function
- **Model-free approach to RL:**
  - ▲ derive the optimal policy without explicitly learning the model
  - ▲ learn an action-based value function or Q-function

## Temporal Difference Learning (TD)

- Based on value iteration, but each step updates the value of only one state: the agent's current state.
- The update is based on a single sample transition rather than the distribution of next states
- For each transition from  $s$  to  $s'$  after taking action  $a$

$$V^\pi(s) = V^\pi(s) + \alpha(R(s,a) + \mathcal{W}^\pi(s') - V^\pi(s))$$

learning rate

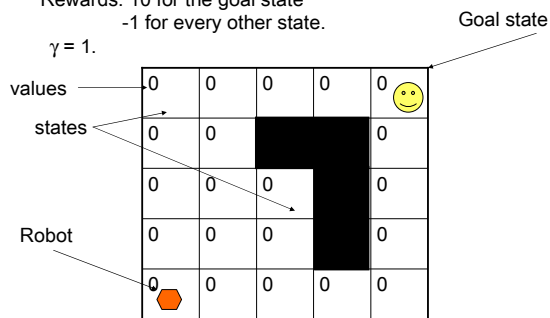
(noisy) sample of utility based on next state

- Intuitively moves us closer to satisfying Bellman constraint

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

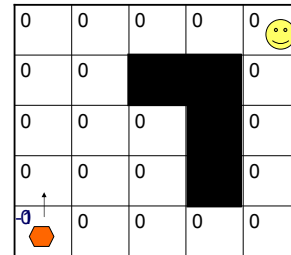
## A Grid Example

Rewards: 10 for the goal state  
-1 for every other state.  
 $\gamma = 1$ .



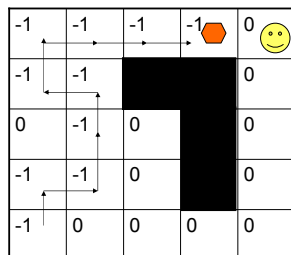
## A Grid Example

Choose an action  $a = \operatorname{argmax}_a (R(s,a) + V(s'))$   
Update  $V(s) \leftarrow \operatorname{Max}_a R(s,a) + V(s')$



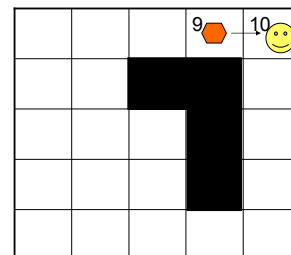
## A Grid Example

Choose an action  $a = \operatorname{argmax}_a R(s,a) + V(s')$   
Update  $V(s) \leftarrow \operatorname{Max}_a R(s,a) + V(s')$



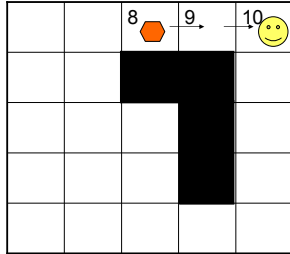
## A Grid Example

Rewards: 10 for the goal state  
-1 for every action



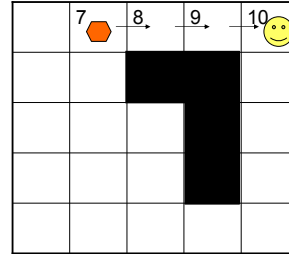
## A Grid Example

Update:  $V(s) \leftarrow \text{Max}_a R(s,a)+V(s')$



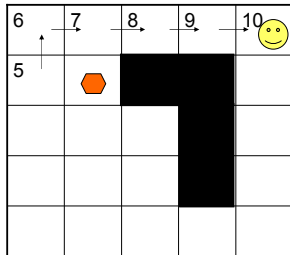
## A Grid Example

Update:  $V(s) \leftarrow \text{Max}_a R(s,a)+V(s')$



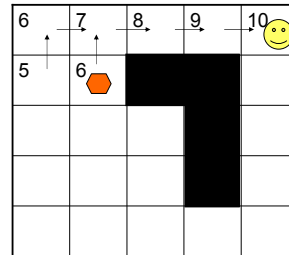
## A Grid Example

Choose an action  $a = \text{argmax}_a R(s,a)+V(s')$   
Update  $V(s) \leftarrow \text{Max}_a R(s,a)+V(s')$

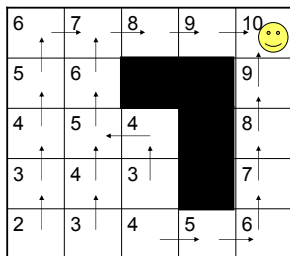


## A Grid Example

Choose an action  $a = \text{argmax}_a R(s,a)+V(s')$   
Update  $V(s) \leftarrow \text{Max}_a R(s,a)+V(s')$



## A Grid Example



The values converge after a few trials *if every action is exercised infinitely often in every state*

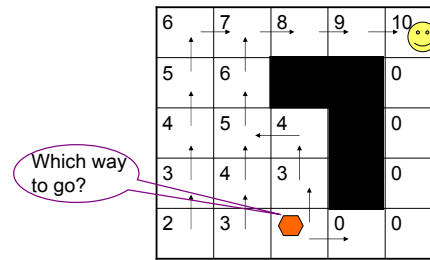
## Comparisons

- 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 similar to direct estimation
  - Partially exploits Bellman constraints--adjusts state to 'agree' with observed successor
    - Not *all* possible successors
  - Convergence in between direct estimation and ADP

## Action Selection

- 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.
- Greedy action** is action maximizing estimated action value
 
$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s')$$
  - where  $V$  is current value function estimate, 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  $s'$
- Always choosing the greedy action (Exploit) could lead the agent to a local optimum.

## Need for Exploration



Always executing the greedy action misses the opportunity to acquire new knowledge. Could converge sub-optimally.

## Explore/Exploit Policies

To guarantee convergence to the optimal policy, we want a policy that is **greedy in the limit of infinite exploration (GLIE)**

- Solution 1:  $\epsilon$ -greedy Exploration**
  - With probability  $1-\epsilon$ , select a greedy action, i.e.,  $\text{argmax}_a Q(s, a)$ . With probability  $\epsilon$  select another random action
  - Decrease  $\epsilon$  with time
- Solution 2: Boltzmann Exploration**
  - Select action  $a$  with probability,
 
$$\Pr(a | s) = \frac{\exp(Q(s, a) / T)}{\sum_{a' \in A} \exp(Q(s, a') / T)}$$
  - $T$  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

## TD-based Policy Optimization

- Start with initial utility/value function
- Take action  $a$  in state  $s$  according to an **explore/exploit policy** (should converge to greedy policy, i.e. GLIE). Let  $s'$  be the next state.
- Update estimated model
- Perform TD update
 
$$V(s) \leftarrow V(s) + \alpha(R(s, a) + \gamma V(s') - V(s))$$

$$V(s)$$
 is new estimate of optimal value function at state  $s$ .
- Goto 2

Requires an estimated model. Why?

To compute  $Q(s, a)$  for greedy policy execution

Can we construct a model-free variant?

## Q-Learning: Model-Free RL

- Instead of learning the optimal value function  $V$ , directly learn the optimal  $Q$  function. Act greedily with respect to  $Q(s, a)$ .
  - Recall that  $Q(s, a)$  is the expected value of taking action  $a$  in state  $s$  and then following the optimal policy thereafter.
- The optimal  $Q$ -function satisfies  $V(s) = \max_a Q(s, a)$  which gives:
 
$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s')$$

$$= R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$$
- After taking action  $a$  in state  $s$  and reaching  $s'$ :
 
$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

(noisy) sample of Q-value based on next state

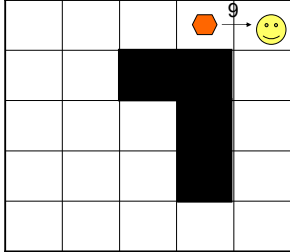
## Q-Learning

- Start with initial  $Q$ -function (e.g. all zeros)
- Take action according to an **explore/exploit policy** (should converge to greedy policy, i.e. GLIE)
- Perform TD update
 
$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$
 $Q(s, a)$  is current estimate of optimal  $Q$ -function.
- Goto 2

- Does not require model since we learn  $Q$  directly!
- Uses explicit  $|S| \times |A|$  table to represent  $Q$
- Explore/exploit policy directly uses  $Q$ -values
  - E.g. use  $\epsilon$ -greedy or Boltzmann exploration.

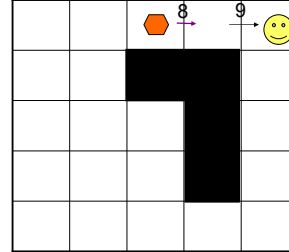
### A Grid Example

Rewards: 10 for reaching the goal state  
 -1 for every action.  $\alpha$  is set to 1 for simplicity.  
 Update:  $Q(s,a) = r + \text{Max}_b (Q(s',b))$



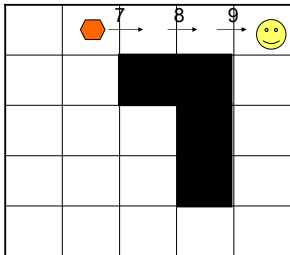
### A Grid Example

Rewards: 10 for reaching the goal state  
 -1 for every action.  $\alpha$  is set to 1 for simplicity.  
 Update:  $Q(s,a) = r + \text{Max}_b (Q(s',b))$



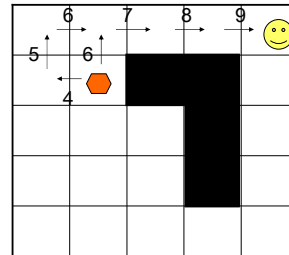
### A Grid Example

Choose an action  $a = \text{argmax}_a Q(s,a)$  reaching  $s'$   
 Update  $Q(s,a) = r + \text{Max}_b Q(s',b)$



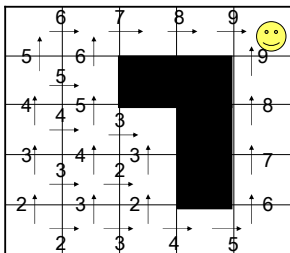
### A Grid Example

Choose an action  $a = \text{argmax}_a Q(s,a)$  reaching  $s'$   
 Update  $Q(s,a) = r + \text{Max}_b Q(s',b)$



### A Grid Example

Choose an action  $a = \text{argmax}_a Q(s,a)$  reaching  $s'$   
 Update  $Q(s,a) = r + \text{Max}_b Q(s',b)$



The values converge to the optimal Q-values under GLIE policy

### Summary

- Dynamic Programming Methods
  - ▲ Value Iteration
  - ▲ Policy Iteration
- Reinforcement Learning Methods
  - ▲ Temporal Difference Learning
  - ▲ Q-learning
- So far we have assumed small state spaces
- Typical state spaces in AI domains are exponentially large -- Bellman's curse of dimensionality

### Value-Based TD vs. Q-learning

- **Value-Based: Learning a model and utility function**
  - ▶ Can be difficult to learn good models for large complex environments  
(e.g. learning a DBN representation)
  - ▶ But if we can learn a model then learning utility function is simpler than learning  $Q(s,a)$
  - ▶ Also can reuse the model for "related problems"
- **Q-learning: Learning Q-function**
  - ▶ Simpler to implement since we don't need to worry about representing and learning a model
  - ▶ But Q-functions can be substantially more complex than utility functions (they must somehow make up for not having the model)