

# Advanced Topics Data Science and AI Automated Planning and Acting

Complex Decision Making

Tanya Braun



UNIVERSITÄT ZU LÜBECK  
INSTITUT FÜR INFORMATIONSSYSTEME

# Content

---

1. Planning and Acting with **Deterministic** Models
2. Planning and Acting with **Refinement** Methods
3. Planning and Acting with **Temporal** Models
4. Planning and Acting with **Nondeterministic** Models
5. Making Simple Decisions
6. Planning and Acting with **Probabilistic** Models
7. Making Complex Decisions
  - a. Markov decision processes (MDP) recap
  - b. Partially-observable MDP (POMDP)
  - c. Dynamic models for online decision making
  - d. *Reinforcement Learning*
8. Provably Beneficial AI
  - Other: open world, perceiving, *learning*
    - If time permits

# Acknowledgements

- Material from Lise Getoor, Jean-Claude Latombe, Daphne Koller, and Stuart Russell, Xiaoli Fern
- Compiled by Ralf Möller
- AIMA Book, Chapters 17 + 21



# Outline

---

## ***Markov decision problem (MDP) recap***

- MDP formalism
- Value iteration, policy iteration

## ***Partially observable Markov decision problem (POMDP)***

- POMDP agent, belief state, belief MDP
- Conditional plans, value iteration

## ***Dynamic graphical models for online decision making***

- Dynamic Bayes nets
- Parameterised dynamic decision models

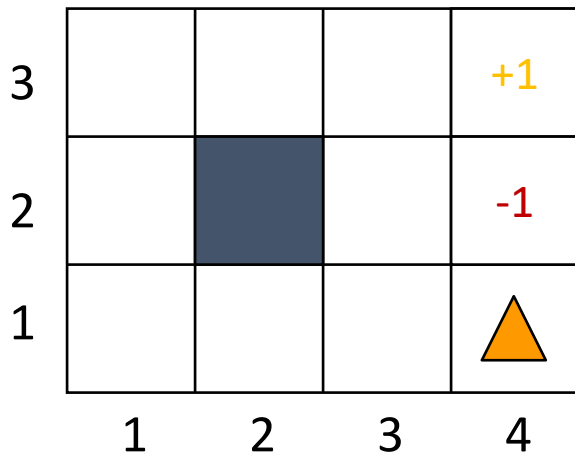
## ***Reinforcement Learning (RL)***

- Active/passive RL
- Model-based/model-free RL
- Multi-armed bandit problem



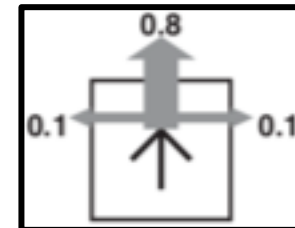
# MDP

- Sequential decision problem for a **fully observable**, **stochastic** environment with a **Markovian transition model** and **additive rewards** (next slide)
- Components
  - a set of states  $S$  (with an initial state  $s_0$ )
  - a set  $A(s)$  of actions in each state
  - a transition model  $P(s'|s, a)$
  - a reward function  $R(s)$



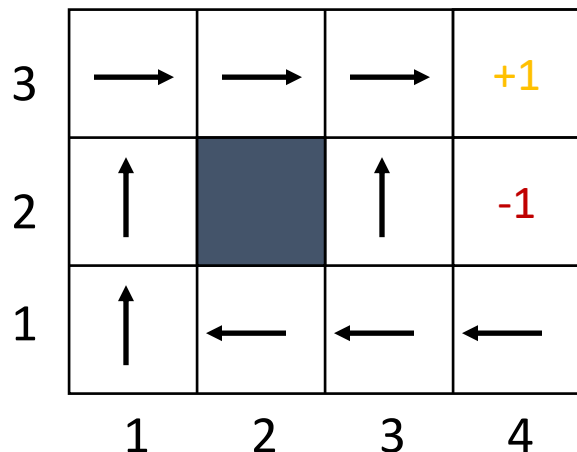
U, D, L, R

each move costs 0.04



# Principle of MEU

- History  $H = (s_0, s_1, \dots, s_n)$ 
  - Utility of  $H$ :  $U(s_0, s_1, \dots, s_n) = \sum_{i=0}^n R(s_i)$
- Bellman equation:
  - $U(s_i) = R(s_i) + \gamma \max_a \sum_{s_j} P(s_j | a, s_i) U(s_j)$
- Optimal policy:
  - $\pi^*(s_i) = \operatorname{argmax}_a \sum_{s_j} P(s_j | a, s_i) U(s_j)$



- Bellman equation for  $[1,1]$

- $U(1,1) = -0.04 + \gamma \max$

$$[ \begin{array}{l} 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), \\ 0.9U(1,1) + 0.1U(1,2), \\ 0.9U(1,1) + 0.1U(2,1), \\ 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1) \end{array} ] .$$

(Up)

(Left)

(Down)

(Right)

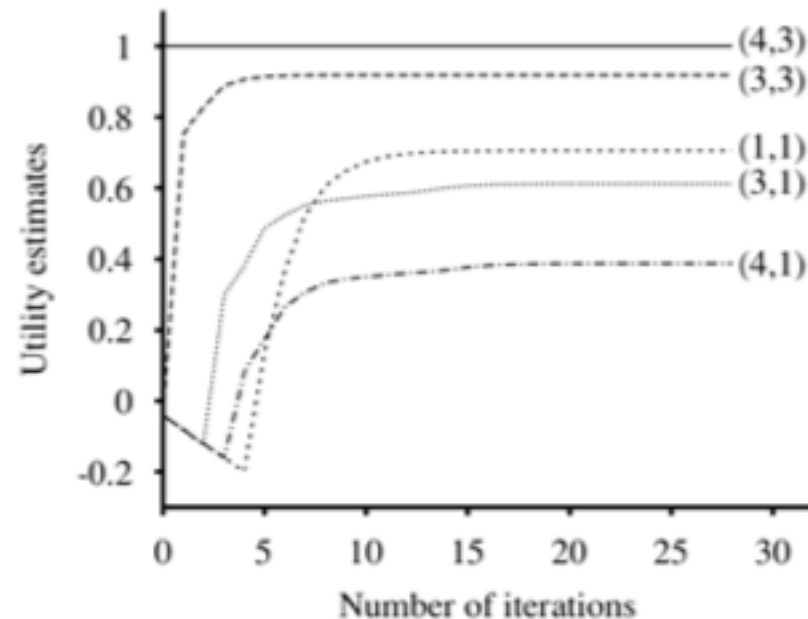
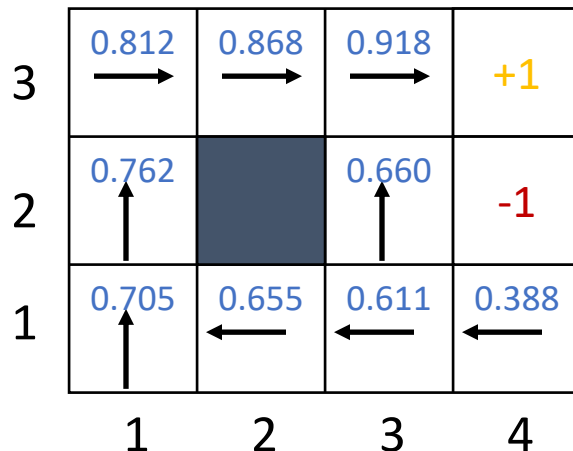
# Value Iteration: Algorithm

```
function value-iteration(mdp,  $\epsilon$ )  
     $U' \leftarrow 0$   
    repeat  
         $U \leftarrow U'$   
         $\delta \leftarrow 0$   
        for each state  $s \in S$  do  
             $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | a, s) U[s']$   
            if  $|U'[s] - U[s]| > \delta$  then  
                 $\delta \leftarrow |U'[s] - U[s]|$   
    until  $\delta < \epsilon(1-\gamma)/\gamma$ 
```

- Inputs
  - an MDP, which includes
    - States  $S$
    - For all  $s \in S$ , actions  $A(s)$ , transition model  $P(s' | a, s)$ , rewards  $R(s)$
    - Discount  $\gamma$
  - Maximum error allowed  $\epsilon$
- Local variables
  - $U, U'$  vectors of utilities for states in  $S$ , initially 0
  - $\delta$  maximum change in utility of any state in an iteration

# Evolution of Utilities

- For  $t = 0, 1, 2, \dots$ , do
  - $U_{t+1}(s_i) = R(s_i) + \gamma \max_a \sum_{s_j} P(s_j | a, s_i) U_t(s_j)$
- Form of information propagation



# Policy Iteration

---

- Pick a policy  $\pi_0$  at random
- Repeat:
  - **Policy evaluation**: Compute the utility of each state for  $\pi_t$ 
    - $U_t(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi_t(s_i), s_i) U_t(s_j)$ 
      - No longer involves a max operation as action is determined by  $\pi_t$
  - **Policy improvement**: Compute the policy  $\pi_{t+1}$  given  $U_t$ 
    - $\pi_{t+1}(s_i) = \arg \max_a \sum_{s_j} P(s_j | \pi_t(s_i), s_i) U_t(s_j)$
  - If  $\pi_{t+1} = \pi_t$ , then return  $\pi_t$

# Intermediate Summary

---

- MDP
  - Markov property
    - Current state depends only on previous state
  - Sequence of actions, history, policy
    - Sequence of actions may yield multiple histories, i.e., sequences of states, with a utility
    - Policy: complete mapping of states to actions
    - Optimal policy: policy with maximum expected utility
  - Value iteration, policy iteration
    - Algorithms for calculating an optimal policy for an MDP

# New Problem

---

- Uncertainty about the world state due to imperfect (partial) information
  - Noise
    - e.g., in sensors
  - Limited accuracy
    - e.g., image resolution, geo-location

# Outline

---

## *Markov decision problem (MDP) recap*

- MDP formalism
- Value iteration, policy iteration

## ***Partially observable Markov decision problem (POMDP)***

- POMDP agent, belief state, belief MDP
- Conditional plans, value iteration

## *Dynamic graphical models for online decision making*

- Dynamic Bayes nets
- Parameterised dynamic decision models

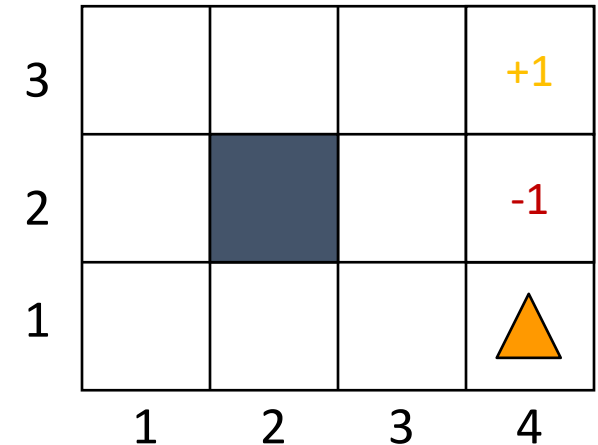
## *Reinforcement Learning (RL)*

- Active/passive RL
- Model-based/model-free RL
- Multi-armed bandit problem

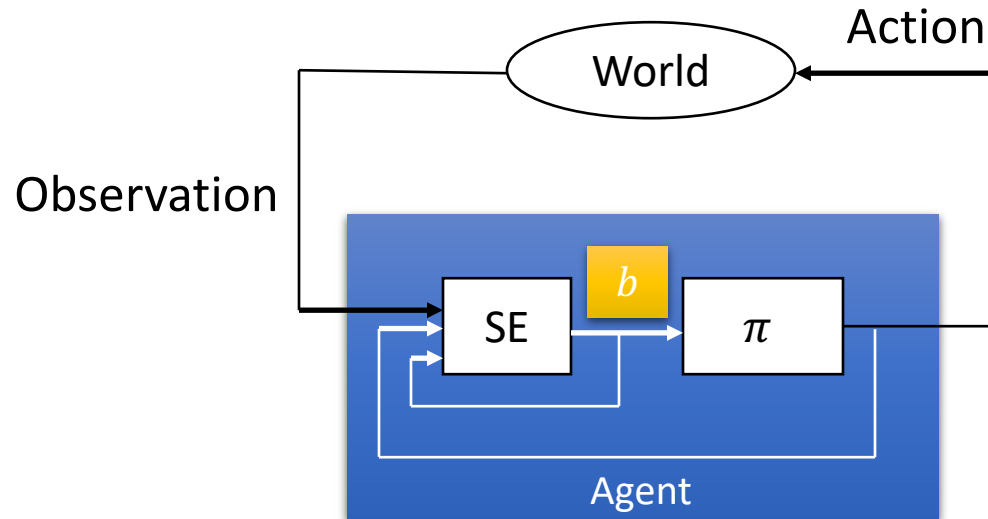


# POMDP

- POMDP = **Partially Observable** MDP
- A sensing operation returns multiple states, with a probability distribution
  - **Sensor model**  $P(o|s)$
  - Example:
    - Sensing number of adjacent walls (1 or 2)
    - Return correct value with probability 0.9
- Choosing the action that maximizes the expected utility of this state distribution assuming “state utilities” computed as before is not good enough, and actually does not make sense (i.e., not rational)
- POMDP agent
  - Constructing a new MDP in which the current probability distribution over states plays the role of the state variable



# Decision cycle of a POMDP agent



- Given the current belief state  $b$ , execute the action  
$$a = \pi^*(b)$$
- Receive observation  $o$
- Set the current belief state to  $SE(b, a, o)$  and repeat
  - SE = State Estimation

# Belief State & Update

- $b(s)$  is the probability assigned to the actual state  $s$  by belief state  $b$
- Update  $b' = SE(b, a, o)$

$$b'(s_j) = P(s_j|o, a, b) = \frac{P(o|s_j, a) \sum_{s_i \in S} P(s_j|s_i, a) b(s_i)}{\sum_{s_k \in S} P(o|s_k, a) \sum_{s_i \in S} P(s_k|s_i, a) b(s_i)}$$

3	0.1	0.1	0.1	0.0
2	0.1		0.1	0.0
1	0.1	0.1	0.1	0.1
	1	2	3	4

- Initial belief state

- Probability of 0 for terminal states
- Uniform distribution for rest
- $b = \left(\frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0\right)$

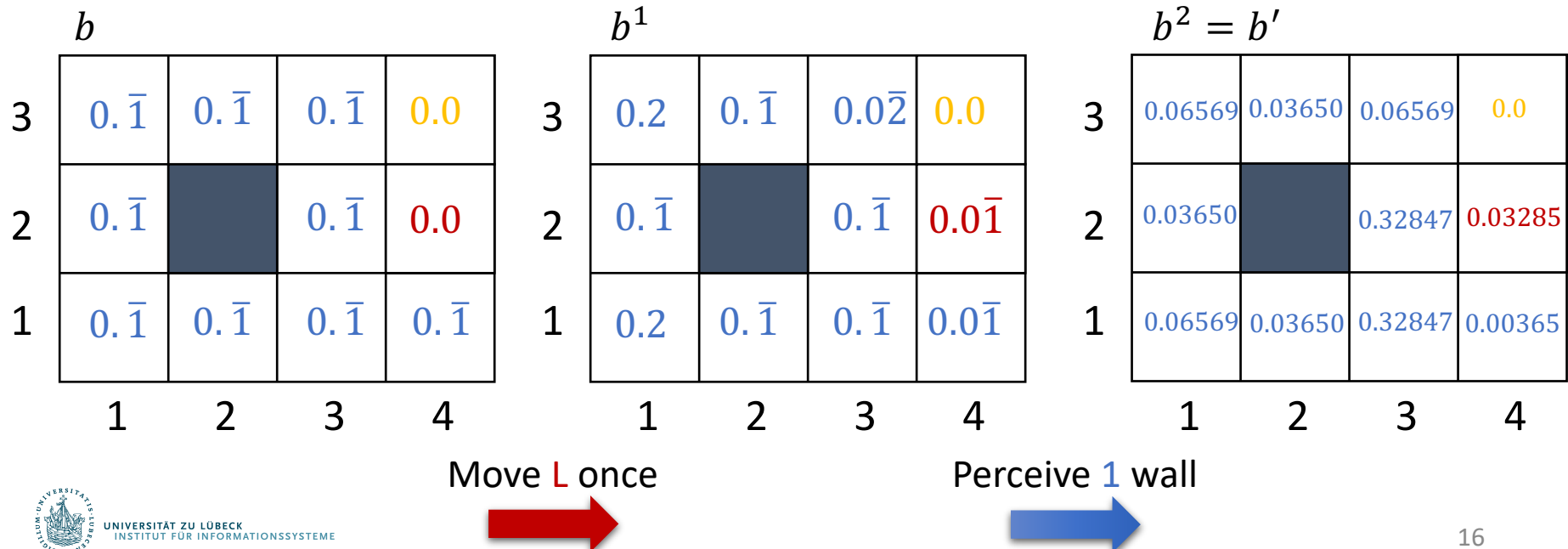
# Belief State & Update

- Update  $b' = SE(b, a, o)$

$$b'(s_j) = P(s_j|o, a, b) = \frac{P(o|s_j, a) \sum_{s_i \in S} P(s_j|s_i, a) b(s_i)}{\sum_{s_k \in S} P(o|s_k, a) \sum_{s_i \in S} P(s_k|s_i, a) b(s_i)}$$

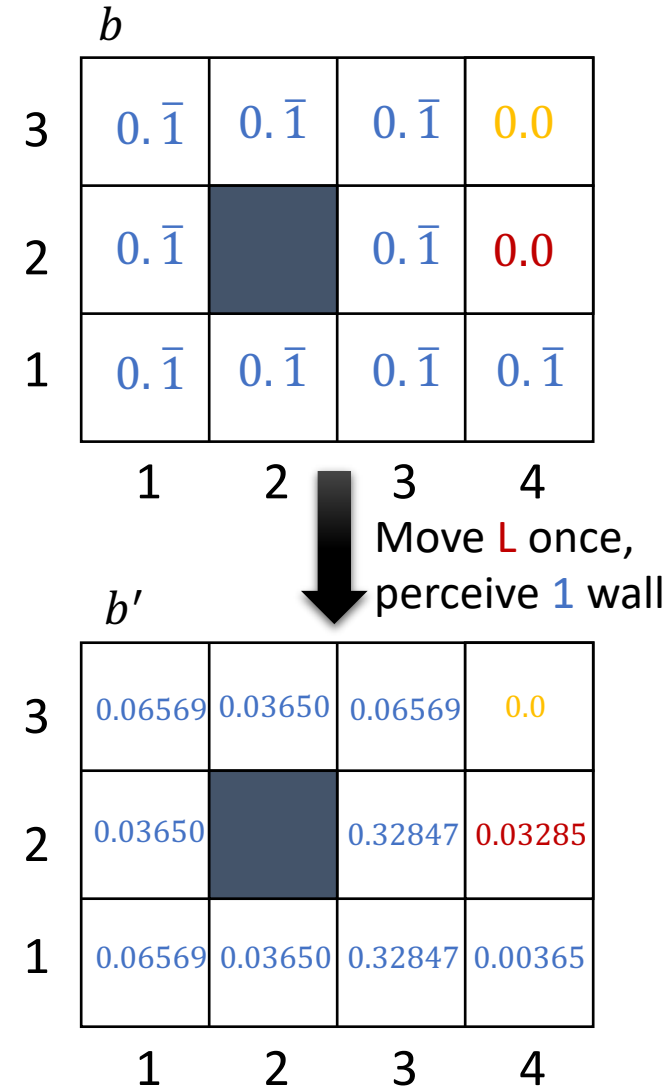
- Consider as two stage-update

1. Update for the **action**
2. Update for the **observation**



# Belief MDP

- A **belief MDP** is a tuple  $(B, A, \rho, P)$ 
  - $B = \text{infinite}$  set of belief states
    - Continuous!
  - $A = \text{finite}$  set of actions
  - Reward function  $\rho(b)$
  - Transition function  $P(b'|b, a)$
  - Sensor model  $P(o|a, b)$



# Belief MDP

- Reward function: Sum over all actual states that the agent can be in

$$\rho(b) = \sum_s b(s)R(s)$$

- Transition function: Sum over all possible observations

$$\begin{aligned} P(b'|b, a) &= \sum_o P(b'|o, a, b)P(o|a, b) \\ &= \sum_o P(b'|o, a, b) \sum_{s'} P(o|s') \sum_s P(s'|s, a)b(s) \end{aligned}$$

- where  $P(b'|o, a, b) = 1$  if  $b' = SE(b, a, o)$  and 0 oth.

- Sensor model: Sum over all actual states that the agent might reach

$$\begin{aligned} P(o|a, b) &= \sum_{s'} P(o|a, s', b)P(s'|a, b) = \sum_{s'} P(o|s')P(s'|a, b) \\ &= \sum_{s'} P(o|s') \sum_s P(s'|s, a)b(s) \end{aligned}$$

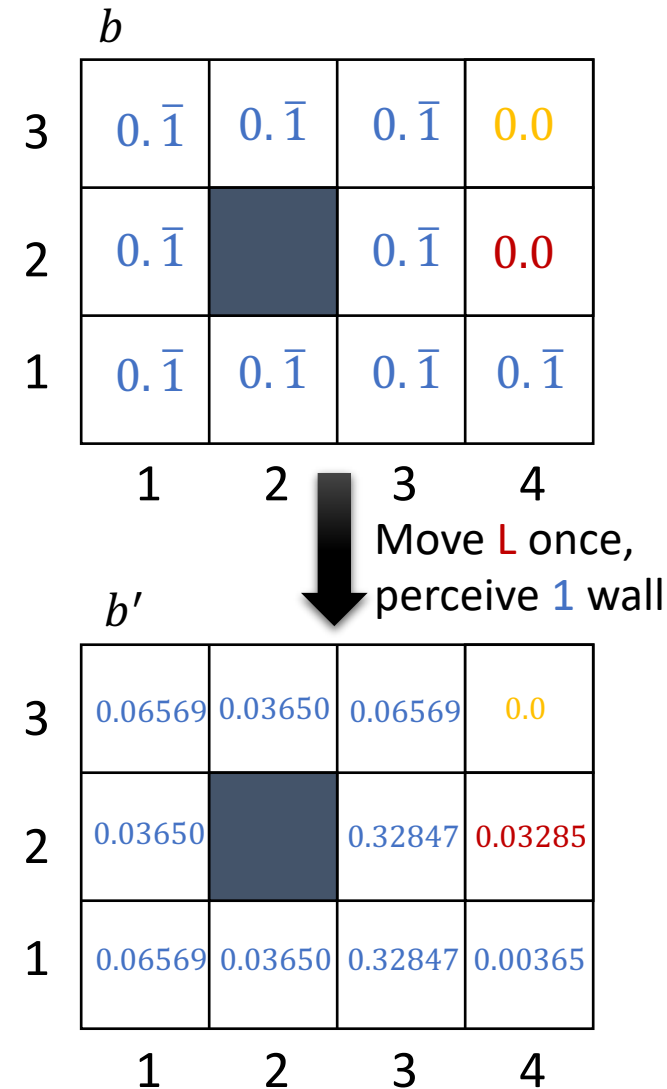
- $P(b'|b, a)$  and  $\rho(b)$  define an **observable MDP** on the **space of belief states**

# Belief MDP

- Optimal action depends only on agent's current belief state
  - Does not depend on actual state the agent is in

⇒ Solving a POMDP on a physical state space is reduced to solving an MDP on the corresponding belief-state space

- Mapping  $\pi^*(b)$  from belief states to actions



# Example Scenario

3	0.111	0.111	0.111	0.000
2	0.111		0.111	0.000
1	0.111	0.111	0.111	0.111
	1	2	3	4

Initial distribution

3	0.300	0.010	0.008	0.000
2	0.221		0.059	0.012
1	0.371	0.012	0.008	0.000
	1	2	3	4

After moving **L** five times

3	0.622	0.221	0.071	0.024
2	0.005		0.003	0.022
1	0.003	0.024	0.003	0.000
	1	2	3	4

After moving **U** five times

3	0.005	0.007	0.019	0.775
2	0.034		0.007	0.105
1	0.005	0.006	0.008	0.030
	1	2	3	4

After moving **R** five times



# Conditional Plans

---

- Example:
  - Two state world 0,1
  - Two actions:  $stay(P)$ ,  $go(P)$ 
    - Actions achieve intended effect with some probability  $P$
  - One-step plan  $[go]$ ,  $[stay]$
- Two-step plans are **conditional**
  - $[a1, \text{IF } percept = 0 \text{ THEN } a2 \text{ ELSE } a3]$
  - Shorthand notation:  $[a1, a2/a3]$
- $n$ -step plans are trees with
  - Nodes attached with actions and
  - Edges attached with percepts

# Value Iteration for POMDPs

- Cannot compute a single utility value for each state of all belief states
- Consider an optimal policy  $\pi^*$  and its application in belief state  $b$
- For this  $b$ , the policy is a **conditional plan**  $p$ 
  - Let the utility of executing a fixed conditional plan  $p$  in  $s$  be  $u_p(s)$
  - Expected utility  $U_p(b) = \sum_s b(s)u_p(s)$ 
    - It varies linearly with  $b$ , a hyperplane in a belief space
  - At any  $b$ , the optimal policy will choose the conditional plan with the highest expected utility

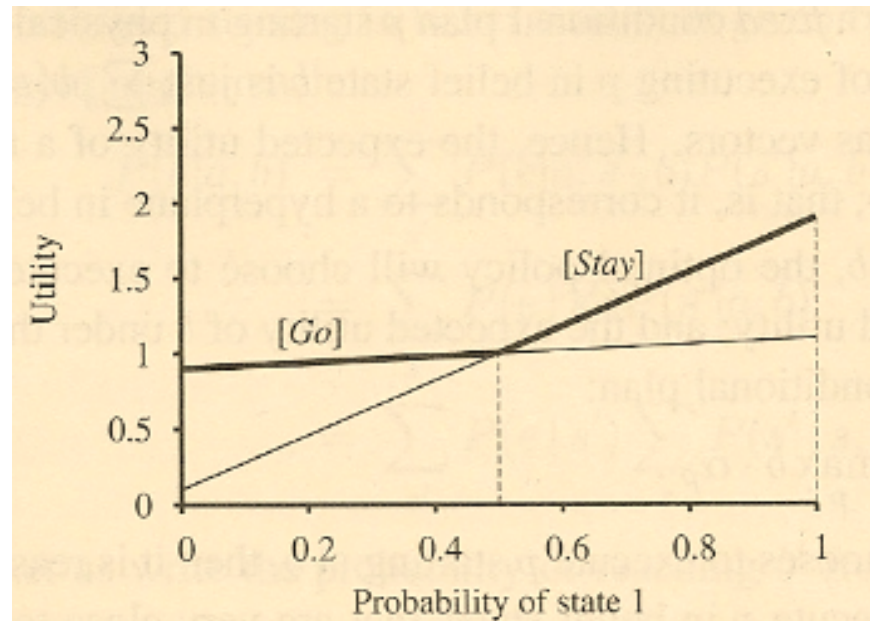
$$U(b) = U^{\pi^*}(b) = \max_p \sum_s b(s)u_p(s)$$
$$\pi^* = \arg \max_p \sum_s b(s)u_p(s)$$

- $U(b)$  is the maximum of a collection of hyperplanes and will be piecewise linear and convex

# Example

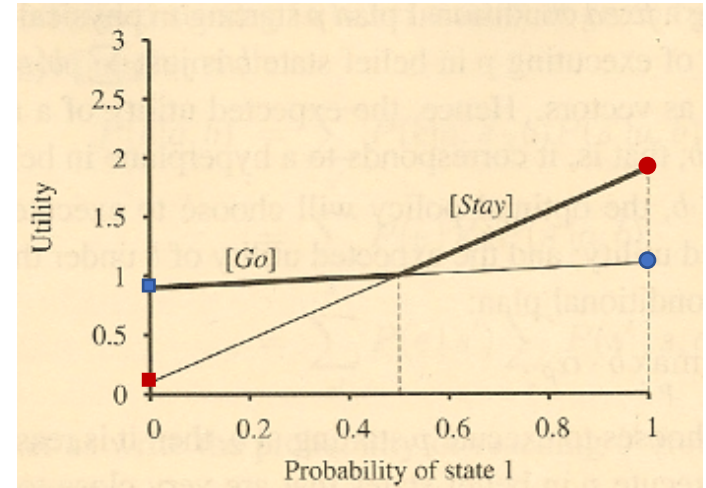
- Compute the utilities for conditional plans of depth 2 by
  - considering each possible first action
  - each possible subsequent percept
  - each way of choosing a depth-1 plan to execute for each percept

Utility of two one-step plans as a function of  $b(1)$



# Example

- Two state world 0,1
- Rewards  $R(0) = 0, R(1) = 1$
- Two actions:  
 $stay(0.9), go(0.9)$
- Sensor reports correct state with probability of 0.6
- Consider the one-step plans  $[stay]$  and  $[go]$ 
  - $u_{[stay]}(0) = R(0) + 0.9R(0) + 0.1R(1) = 0.1$  ■
  - $u_{[stay]}(1) = R(1) + 0.9R(1) + 0.1R(0) = 1.9$  •
  - $u_{[go]}(0) = R(0) + 0.9R(1) + 0.1R(0) = 0.9$  ■
  - $u_{[go]}(1) = R(1) + 0.9R(0) + 0.1R(1) = 1.1$  •
  - This is just the direct reward function (taken into account the probabilistic transitions)



# Example

- 8 distinct depth-2 plans for each state

Utility of depth-1 plan  
given state, outcome of  
first action, and percept

After moving from  
0 to 1, perceive  
false state (0);  
plan says *stay* for  
0, so receive  
 $u_{[stay]}(1) = 1.9$

Sum over states reachable  
with first action

Probability of  
next state

Probability of  
correct percept

$$\begin{aligned}
 u_{[stay, stay/stay]}(0) &= R(0) + (0.9 \cdot (0.6 \cdot 0.1 + 0.4 \cdot 0.1) + 0.1 \cdot (0.6 \cdot 1.9 + 0.4 \cdot 1.9)) = 0.28 \\
 u_{[stay, stay/stay]}(1) &= R(1) + (0.9 \cdot (0.6 \cdot 1.9 + 0.4 \cdot 1.9) + 0.1 \cdot (0.6 \cdot 0.1 + 0.4 \cdot 0.1)) = 2.72
 \end{aligned}$$

Reward of state

Sum over possible percepts

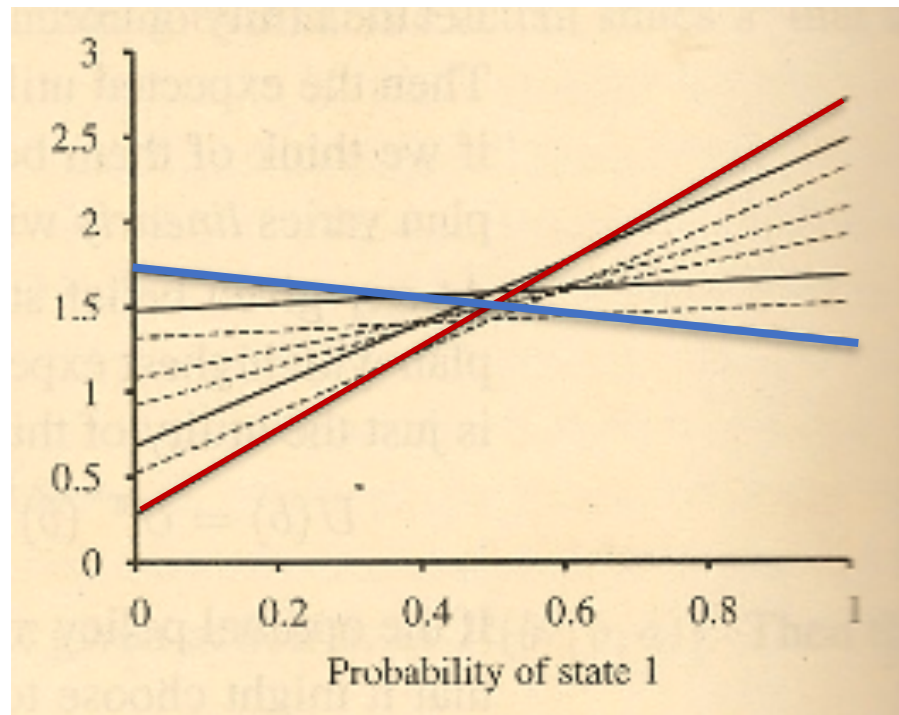
$$\begin{aligned}
 &u_{[stay, go/stay]}(0), u_{[stay, stay/go]}(0), u_{[stay, go/go]}(0) \\
 &u_{[stay, go/stay]}(1), u_{[stay, stay/go]}(1), u_{[stay, go/go]}(1)
 \end{aligned}$$

$$\begin{aligned}
 u_{[go, stay/stay]}(0) &= R(0) + (0.9 \cdot (0.6 \cdot 1.9 + 0.4 \cdot 1.9) + 0.1 \cdot (0.6 \cdot 0.1 + 0.4 \cdot 0.1)) = 1.72 \\
 u_{[go, stay/stay]}(1) &= R(1) + (0.9 \cdot (0.6 \cdot 0.1 + 0.4 \cdot 0.1) + 0.1 \cdot (0.6 \cdot 1.9 + 0.4 \cdot 1.9)) = 1.28
 \end{aligned}$$

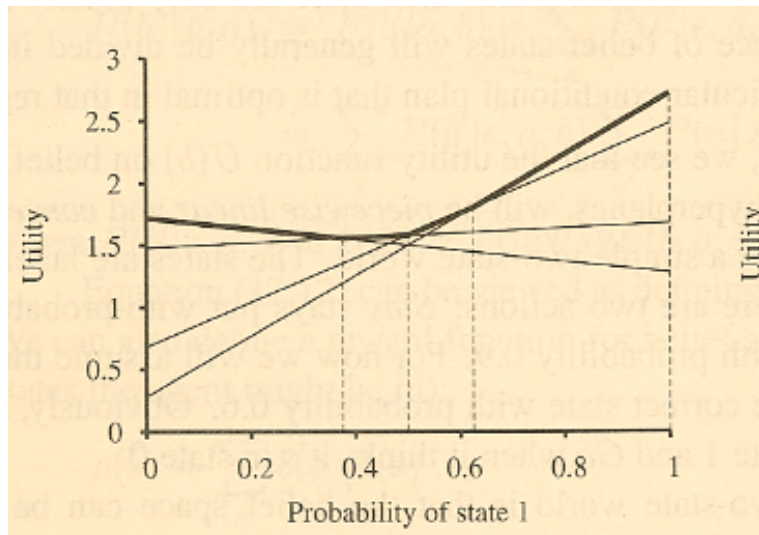
$$\begin{aligned}
 &u_{[go, go/stay]}(0), u_{[go, stay/go]}(0), u_{[go, go/go]}(0) \\
 &u_{[go, go/stay]}(1), u_{[go, stay/go]}(1), u_{[go, go/go]}(1)
 \end{aligned}$$

# Example

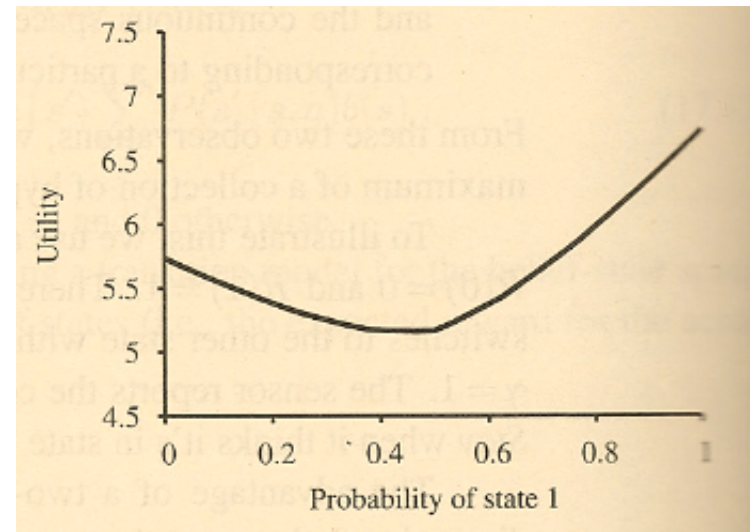
- 8 distinct depth-2 plans for state 1
  - 4 are suboptimal across the entire belief space (dashed lines)
  - With probability  $b(1) = 1$ :
    - $u_{[stay,stay/stay]}(0) = 0.28$   $u_{[stay,stay/stay]}(1) = 2.72$
    - $u_{[go,stay/stay]}(0) = 1.72$   $u_{[go,stay/stay]}(1) = 1.28$



# Example



Utility of four undominated two-step plans



Utility function for optimal eight step plans

# General formula

- Let  $p$  be a depth- $d$  conditional plan whose initial action is  $a$  and whose depth- $d - 1$  subplan for percept  $e$  is  $p.e$ , then

$$u_p(s) = R(s) + \sum_{s'} P(s' | s, a) \sum_e P(e | s') u_{p.e}(s')$$

- This gives us a *value iteration* algorithm
- The elimination of dominated plans is essential for reducing doubly exponential growth:
  - Number of undominated plans with  $d = 8$  is just 144
  - Otherwise  $2^{255} (|A|^{O(|E|^{d-1})})$ 
    - For large POMDPs this approach is highly inefficient



# Value Iteration: Algorithm

```
function value-iteration(pomdp,  $\epsilon$ )
```

```
   $U' \leftarrow$  a set containing the empty plan [] with  $u_{[]} (s) = R(s)$ 
```

```
  repeat
```

```
     $U \leftarrow U'$ 
```

```
     $U \leftarrow$  the set of all plans consisting of an action and,  
                for each possible next percept, a plan in  $U$  with  
                utility vectors computed as on previous slide
```

```
     $U' \leftarrow \text{Remove-dominated-plans}(U')$ 
```

```
until  $\text{Max-difference}(U, U') < \epsilon(1-\gamma)/\gamma$ 
```

```
return  $U$ 
```

- Inputs

- a POMDP, which includes

- States  $S$
- For all  $s \in S$ , actions  $A(s)$ , transition model  $P(s' | a, s)$ , sensor model  $P(o | s)$ , rewards  $\rho(s)$
- Discount  $\gamma$

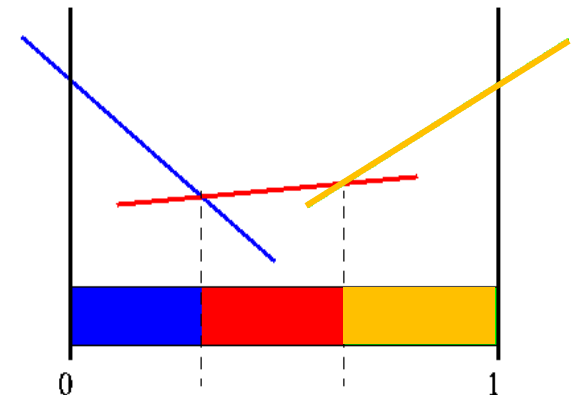
- Maximum error allowed  $\epsilon$

- Local variables

- $U, U'$  sets of plans with associated utility vectors  $u_p$

# Solutions for POMDP

- Belief MDP has reduced POMDP to MDP
  - MDP obtained has a multidimensional continuous state space
- Extract a policy from utility function returned by value-iteration algorithm
  - Policy  $\pi(b)$  can be represented as a set of **regions** of belief state space
  - Each region associated with a particular optimal action
  - Value function associates distinct linear function of  $b$  with each region
  - Each value or policy iteration step refines the boundaries of the regions and may introduce new regions.



# Intermediate Summary

---

- POMDP
  - Uncertainty about state  $\rightarrow$  belief state
  - Solving a POMDP = Solving an MDP on space of belief states
  - Policy = conditional plans
  - Value iteration to find optimal policy
    - Very expensive, even with deletion of dominated plans
  - Offline

# Outline

---

## *Markov decision problem (MDP) recap*

- MDP formalism
- Value iteration, policy iteration

## *Partially observable Markov decision problem (POMDP)*

- POMDP agent, belief state, belief MDP
- Conditional plans, value iteration

## ***Dynamic graphical models for online decision making***

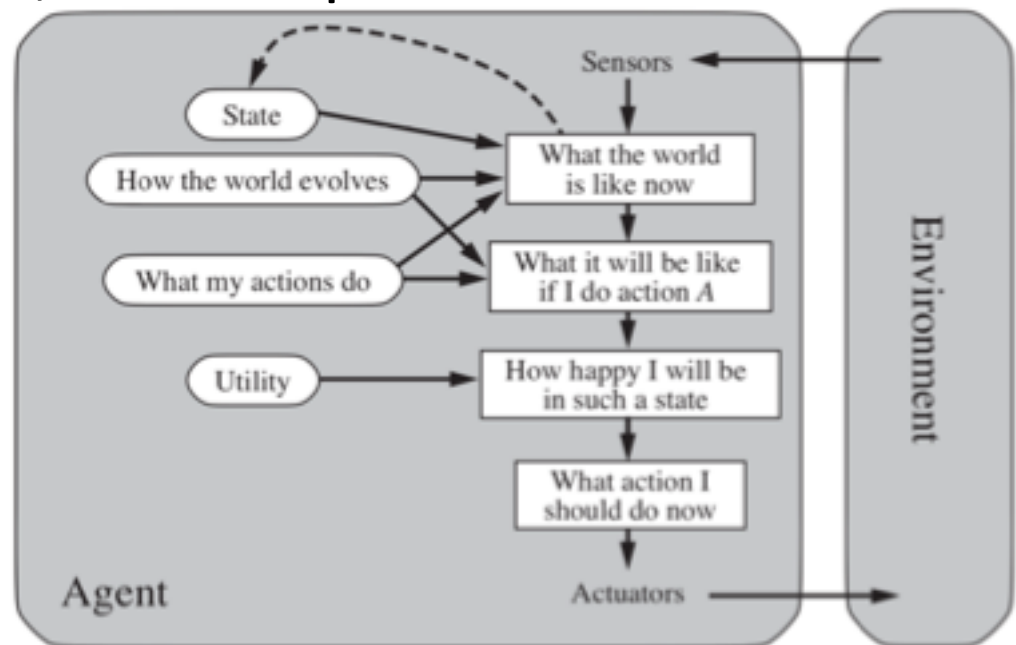
- Dynamic Bayes nets
- Parameterised dynamic decision models

## *Reinforcement Learning (RL)*

- Active/passive RL
- Model-based/model-free RL
- Multi-armed bandit problem

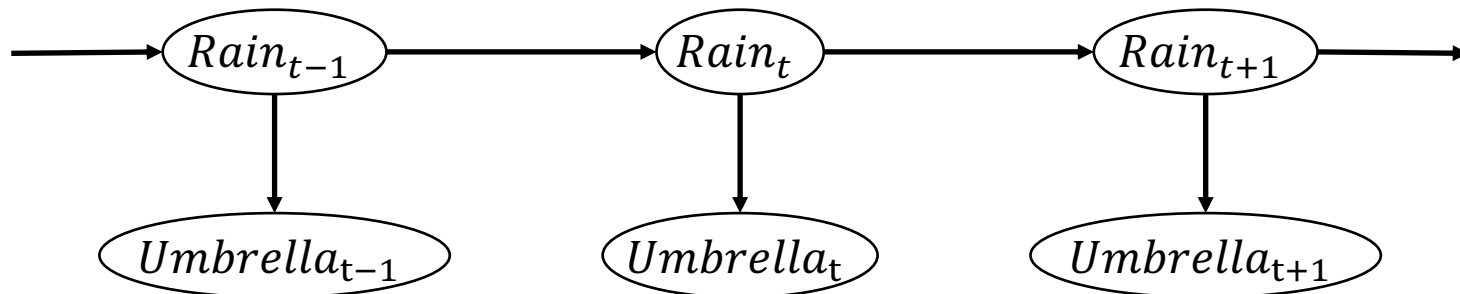
# Online Agents for POMDPs

- Transition and sensor models represented as a dynamic graphical model, extended with actions/decisions and utilities
  - Dynamic decision network
- Inference algorithm (filtering) to incorporate each new percept, action, and to update the belief state representation
- Decisions made by projecting forward possible action sequences and choosing the best one (MEU)



# Dynamic Graphical Models

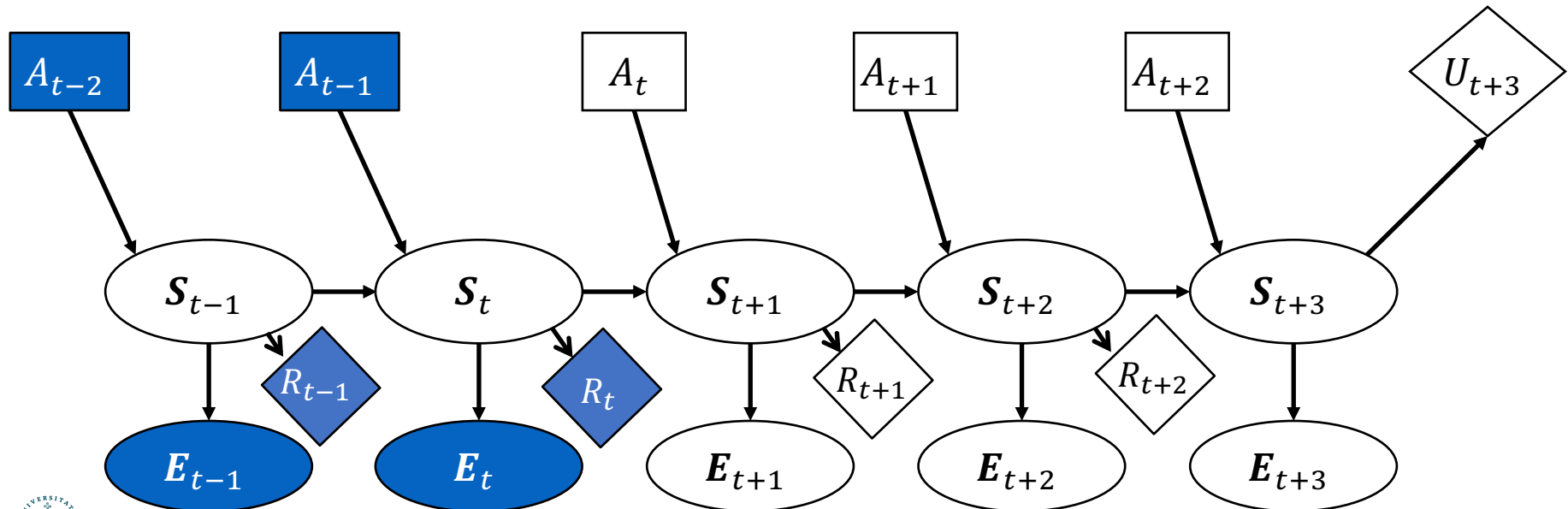
- Dynamic = sequential or temporal
  - Series of snapshots, indexed by  $t$
- Models to extend with actions and utilities
  - **Dynamic Bayesian network**
    - Directed, conditional probability tables, propositional
    - Special case: HMM
      - DBN with only one state and one evidence variable, Markov-1



- **Dynamic parameterised models**
  - Undirected, factors, relational domains

# Dynamic Bayesian Decision Networks

- The decision problem involves calculating the value of  $A_t$  that maximizes the agent's expected utility over the remaining state sequence
  - Blue nodes: observed/known
  - Current timestep  $t$
  - Finite horizon of 3 (Lookahead)



## 36



# Discussion of DDNs

---

- DDNs provide a general, concise representation for large POMDPs
- Agent systems moved from
  - static and simple environments to
  - dynamic and complex environments that are closer to the real world
- However, exact algorithms are exponential in tree width

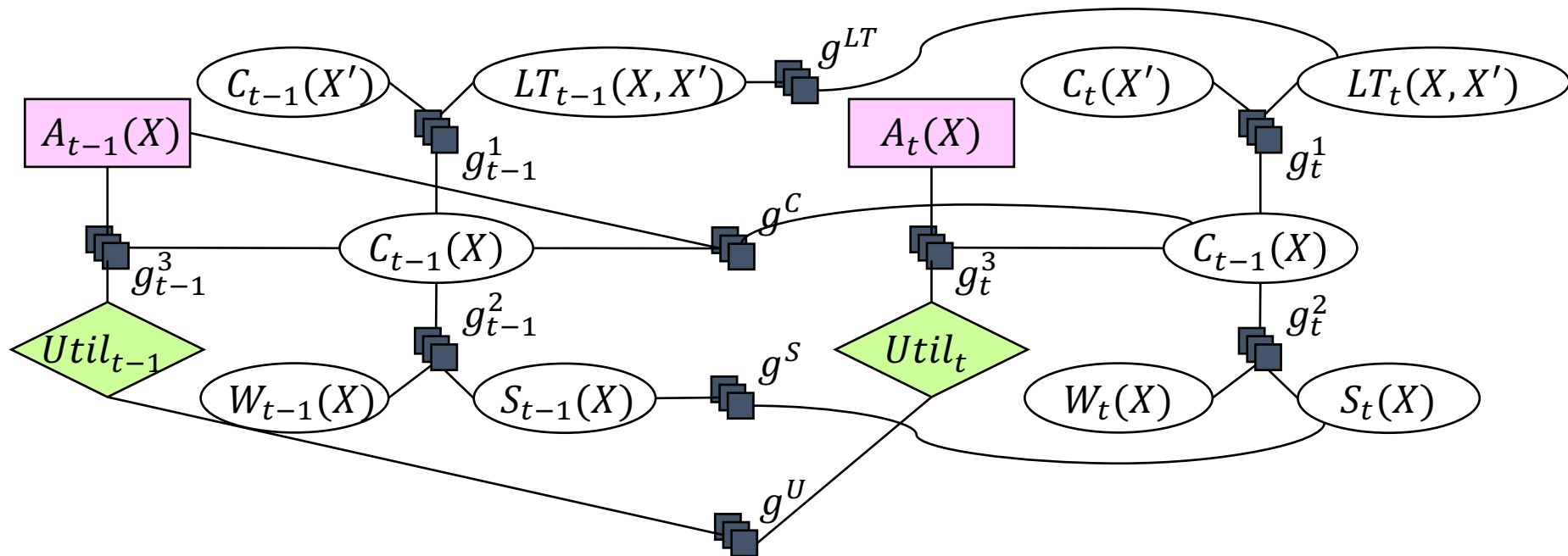
# Perspectives to Reduce Complexity

---

- Combined with a heuristic estimate for the utility of the remaining steps
- Incremental pruning techniques
- Many approximation techniques:
  - Using less detailed state variables for states in the distant future
  - Using a greedy heuristic search through the space of decision sequences
  - Assuming “most likely” values for future percept sequences rather than considering all possible values
- Parameterised models for relational domains

# Dynamic Parameterised Model

- Relational model
  - Allows for lifted calculations
- For decision making
  - With actions and utilities



# Dynamic Parameterised Decisions

---

- Online inference (as before)
  - Repeat
    - Update current belief state to percepts (evidence)
    - Calculate best action for next timestep based on the projected expected utility  $k$  timesteps into the future
    - Perform best action
  - Finite horizon ( $k$ )
- Calculations using lifted inference operators
  - Same decision/action for groups of indistinguishable constants
  - Polynomial with respect to domain sizes
    - Worst case = propositional case

# Intermediate Summary

---

- Dynamic graphical models
  - Complex representation of environment
  - Sequential decision making
    - Online
    - Limited horizon
    - Exact algorithms exponential in tree width
  - Relational models for complexity polynomial with respect to domain size

# Outline

---

## *Markov decision problem (MDP)*

- MDP formalism
- Value iteration, policy iteration

## *Partially observable Markov decision problem (POMDP)*

- POMDP agent, belief state, belief MDP
- Conditional plans, value iteration

## *Dynamic graphical models for online decision making*

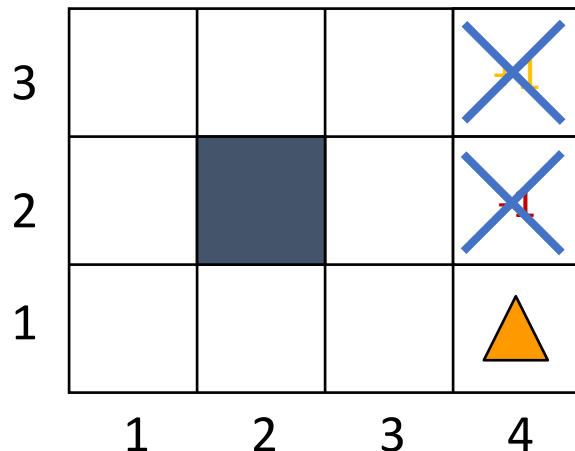
- Dynamic Bayes nets
- Parameterised dynamic decision models

## ***Reinforcement Learning (RL)***

- Active/passive RL
- Model-based/model-free RL
- Multi-armed bandit problem

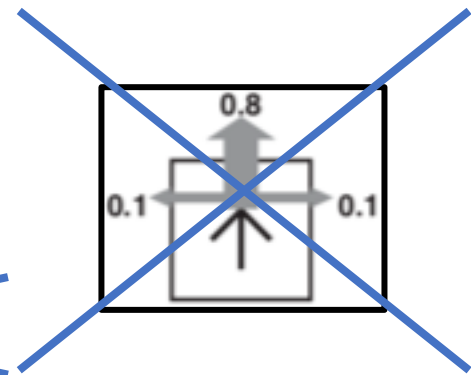
# Reinforcement Learning (RL)

- Agent placed in an environment and must learn to behave optimally in it
- Assume that the world behaves like an MDP, except
  - Agent can act but does not know the transition model
  - Agent observes its current state and its reward but does not know the reward function
- Goal: learn an optimal policy



U, D, L, R

each move costs 0.04



# Factors that make RL

---

- Actions have non-deterministic effects
  - which are initially unknown and must be learned
- Rewards / punishments can be 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 problem
  - World is large and complex



# Passive vs. Active Learning

---

- **Passive** learning
  - Agent acts based on a fixed policy  $\pi$  and tries to learn how good the policy is by observing the world go by
  - Analogous to policy iteration
- **Active** learning
  - Agent attempts to find an optimal (or at least good) policy by exploring different actions in the world
  - Analogous to solving the underlying MDP

# Model-based vs. Model-free RL

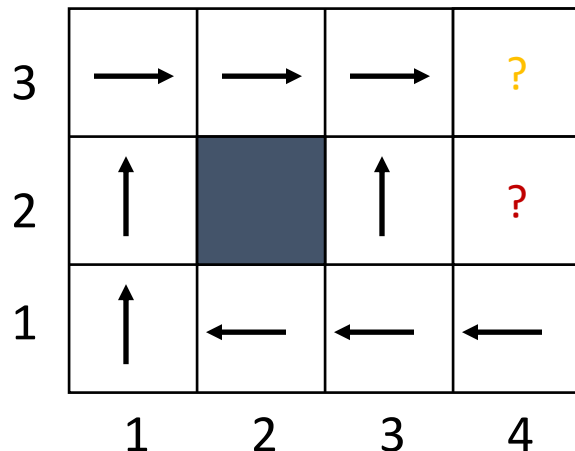
---

- **Model-based** approach to RL
  - Learn the MDP model ( $P(s'|s, a)$  and  $R$ ), or an approximation of it
  - Use it to find the optimal policy
- **Model-free** approach to RL
  - Derive the optimal policy without explicitly learning the model

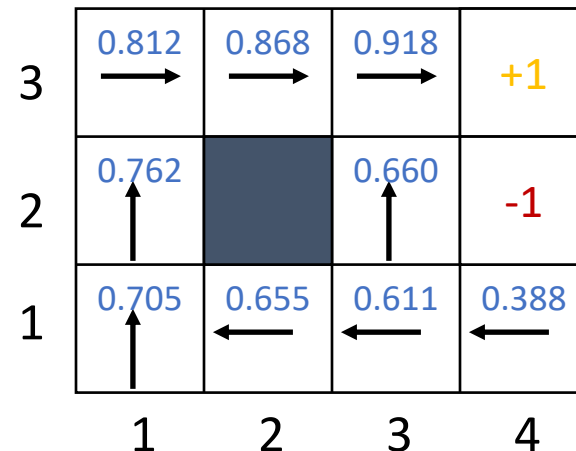
# Passive RL

- Suppose we are given a policy
- Want to determine how good it is

- Given  $\pi$ :



Need to learn  $U^\pi(s)$ :



# Passive RL

- Given policy  $\pi$ :
  - Estimate  $U^\pi(s)$
- Not given
  - Transition model  $P(s'|s, a)$
  - Reward function  $R(s)$
- Simply follow the policy for many **epochs**
- Epochs: training sequences

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

(1,1) → (1,2) → (1,3) → (1,2) → (1,3) → (2,3) → (3,3) → (3,4) + 1  
(1,1) → (1,2) → (1,3) → (2,3) → (3,3) → (3,2) → (3,3) → (3,4) + 1  
(1,1) → (2,1) → (3,1) → (3,2) → (4,2) - 1

# Direct Utility Estimation (DUE)

---

- Model-free approach
  - Estimate  $U^\pi(s)$  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

# DUE: Example

- Suppose we observe the following trial:
  - $(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow$   
 $(1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,4)_{+1}$
- The total reward starting at  $(1,1)$  is 0.72
  - Call this a sample of the observed-reward-to-go for  $(1,1)$
- For  $(1,2)$ , there are two samples for the observed-reward-to-go (assuming  $\gamma = 1$ )
  1.  $(1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow$   
 $(2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,4)_{+1}$  [Total: 0.76]
  2.  $(1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow$   
 $(3,4)_{+1}$  [Total: 0.84]

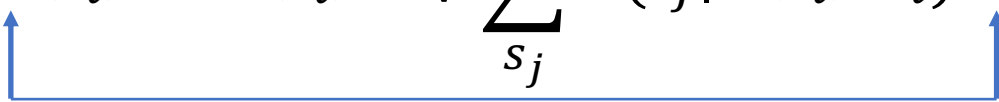
# DUE: Convergence

---

- Keep a running average of the observed reward-to-go for each state
  - E.g., for state (1,2), it stores  $\frac{(0.76+0.84)}{2} = 0.8$
- As the number of trials goes to infinity, the sample average converges to the true utility

# DUE: Problem

- Big problem: **it converges very slowly!**
- Why?
  - Does not exploit the fact that utilities of states are not independent
  - Utilities follow the Bellman equation

$$U^\pi(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi(s_i), s_i) U^\pi(s_j)$$


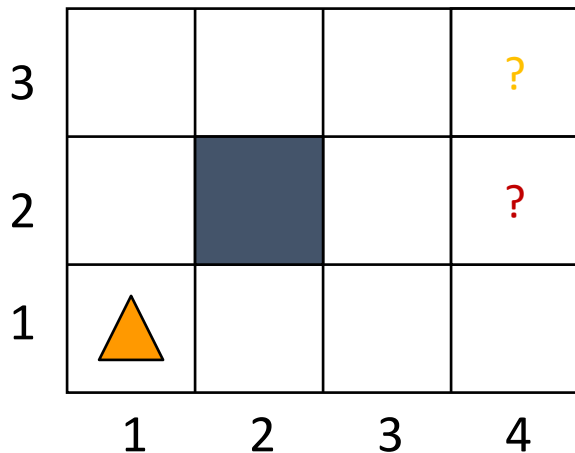
Dependence on neighbouring states

The diagram consists of a blue horizontal line with two vertical arrows pointing upwards from its ends. The left arrow points to the term  $U^\pi(s_i)$  in the equation above, and the right arrow points to the term  $U^\pi(s_j)$ . This visualizes the recursive dependency of the utility of state  $s_i$  on the utility of its successor state  $s_j$ .



# DUE: Problem

- Using the dependence to your advantage
  - Suppose you know that state  $(3,3)$  has a high utility
  - Suppose you are now at  $(3,2)$
  - Bellman equation would be able to tell you that  $(3,2)$  is likely to have a high utility because  $(3,3)$  is a neighbour
- DUE cannot tell you that until the end of the trial



# Adaptive Dynamic Programming (ADP)

---

- Model-based approach
- Basically learns the transition model  $P(s'|s, a)$  and the reward function  $R(s)$ 
  - Takes advantage of constraints in the Bellman equation
- Given policy  $\pi$ :
  - Estimate  $U^\pi(s)$
  - Based on  $P(s'|s, a)$  and  $R(s)$ , we can perform policy evaluation (part of policy iteration)

# ADP: Policy Evaluation

- Policy Iteration:

- Pick a policy  $\pi$  at random
- Repeat:

- **Policy evaluation:** Compute the utility of each state for  $\pi$

- $U_{t+1}(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi(s_i), s_i) U_t(s_j)$

- No longer involves a max operation as action is determined by  $\pi$

- Policy improvement: Compute the policy  $\pi'$  given  $U_{t+1}$

- $\pi'(s_i) = \arg \max_a \sum_{s_j} P(s_j | \pi(s_i), s_i) U_t(s_j)$

- If  $\pi' = \pi$ , then return  $\pi$

Can be solved in time  $O(n^3)$ , where  $n$  is the number of states

Or solve the set of linear equations:

$$U(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi(s_i), s_i) U(s_j)$$

(often a sparse system)

# ADP: Learn the Model

---

- Make use of policy evaluation to learn the utilities of states

- To use policy equation

$$U_{t+1}(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi(s_i), s_i) U_t(s_j)$$

agent needs to learn  $P(s'_j | s, a)$  and  $R(s)$

- How?

# ADP: Learn the Model

---

- Learning  $R(s)$ 
  - Easy because it is deterministic
  - Whenever you see a new state, store the observed reward value as  $R(s)$
- Learning  $P(s'|s, a)$ 
  - Keep track of how often you get to state  $s'$  given that you are in state  $s$  and do action  $a$
  - E.g., if you are in  $s = (1,3)$  and you execute **R** three times and you end up in  $s' = (2,3)$  twice, then
$$P(s'|\mathbf{R}, s) = \frac{2}{3}$$

# ADP: Algorithm

Update  
reward  
function

Update  
transition  
model

```
function passive-ADP-agent(percept)
  returns an action
  input: percept, indicating current state  $s'$ , reward  $r'$ 
  static:
     $\pi$ , fixed policy
    mdp, MDP with  $P[s'|s,a]$ ,  $R(s)$ ,  $\gamma$ 
     $U$ , table of utilities, initially empty
     $N_{sa}$ , table of freq. for  $s$ - $a$  pairs, initially 0
     $N_{sas'}$ , table of freq. for  $s$ - $a$ - $s'$  triples, initially 0
     $s, a$ , previous state and action, initially null

  if  $s'$  is new then
     $U[s'] \leftarrow r'$ 
     $R[s'] \leftarrow r'$ 

  if  $s$  is not null then
    increment  $N_{sa}[s,a]$  and  $N_{sas'}[s,a,s']$ 
    for each  $t$  s.t.  $N_{sas'}[s,a,t] \neq 0$  do
       $P[t|s,a] \leftarrow N_{sas'}[s,a,t] / N_{sa}[s,a]$ 
     $U \leftarrow \text{Policy-evaluation}(\pi, U, \text{mdp})$ 
  if Terminal?( $s'$ ) then
     $s, a \leftarrow \text{null}$ 
  else
     $s, a \leftarrow s', \pi[s']$ 
  return  $a$ 
```

# ADP: Problem

---

- Need to solve a system of simultaneous equations – costs  $O(n^3)$ 
  - Very hard to do if you have  $10^{50}$  states like in Backgammon
  - Could make things a little easier with modified policy iteration
- Can we avoid the computational expense of full policy evaluation?

# Temporal Difference Learning (TD)

---

- Instead of calculating the exact utility for a state, can we approximate it and possibly make it less computationally expensive?

- Yes, we can! Using TD:

$$U^{\pi}(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi(s_i), s_i) U^{\pi}(s_j)$$

- Instead of doing the sum over all successors, only adjust the utility of the state based on the successor observed in the trial
- Does not estimate the transition model – model-free



# TD: Example

---

- Suppose you see that  $U^\pi(1,3) = 0.84$  and  $U^\pi(2,3) = 0.92$
- If the transition  $(1,3) \rightarrow (2,3)$  happens all the time, you would expect to see:
$$U^\pi(1,3) = R(1,3) + U^\pi(2,3)$$
$$\Rightarrow U^\pi(1,3) = -0.04 + U^\pi(2,3)$$
$$\Rightarrow U^\pi(1,3) = -0.04 + 0.92 = 0.88$$
- Since you observe  $U^\pi(1,3) = 0.84$  in the first trial and it is a little lower than 0.88, so you might want to “bump” it towards 0.88

# Aside: Online Mean Estimation

- Suppose that we want to incrementally compute the mean of a sequence of numbers
  - E.g., to estimate the mean of a random variable from a sequence of samples

average of  $n + 1$  samples

$$\begin{aligned}\hat{X}_{n+1} &= \frac{1}{n+1} \sum_{i=1}^{n+1} x_i = \left( \frac{1}{n+1} \sum_{i=1}^n x_i \right) + \frac{1}{n+1} x_{n+1} = \left( \frac{n}{n(n+1)} \sum_{i=1}^n x_i \right) + \frac{1}{n+1} x_{n+1} \\ &= \left( \frac{n+1-1}{n(n+1)} \sum_{i=1}^n x_i \right) + \frac{1}{n+1} x_{n+1} = \left( \frac{n+1}{n(n+1)} \sum_{i=1}^n x_i \right) - \left( \frac{1}{n(n+1)} \sum_{i=1}^n x_i \right) + \frac{1}{n+1} x_{n+1} \\ &= \left( \frac{1}{n} \sum_{i=1}^n x_i \right) - \left( \frac{1}{(n+1)} \cdot \frac{1}{n} \sum_{i=1}^n x_i \right) + \frac{1}{n+1} x_{n+1} = \left( \frac{1}{n} \sum_{i=1}^n x_i \right) + \frac{1}{n+1} \left( x_{n+1} - \frac{1}{n} \sum_{i=1}^n x_i \right) \\ &= \hat{X}_n + \frac{1}{n+1} (x_{n+1} - \hat{X}_n)\end{aligned}$$

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

# TD Update

- TD update for transition from  $s$  to  $s'$

$$U^\pi(s) = U^\pi(s) + \alpha \left( \underbrace{R(s) + \gamma U^\pi(s')}_{\text{new (noisy) sample of utility based on next state}} - U^\pi(s) \right)$$

learning rate

new (noisy) sample of utility  
based on next state

- Similar to one step of value iteration
- Equation called **backup**
- So, the update is maintaining a “mean” of the (noisy) utility samples
- If the learning rate decreases with the number of samples (e.g.,  $1/n$ ), then the utility estimates will eventually converge to true values

$$U^\pi(s_i) = R(s_i) + \gamma \sum_{s_j} P(s_j | \pi(s_i), s_i) U^\pi(s_j)$$

# TD: Convergence

---

- Since we are using the observed successor  $s'$  instead of all the successors, what happens if the transition  $s \rightarrow s'$  is very rare and there is a big jump in utilities from  $s$  to  $s'$ ?
  - How can  $U^\pi(s)$  converge to the true equilibrium value?
- Answer: The average value of  $U^\pi(s)$  will converge to the correct value
- This means we need to observe enough trials that have transitions from  $s$  to its successors
- Essentially, the effects of the TD backups will be averaged over a large number of transitions
- Rare transitions will be rare in the set of transitions observed

# Comparison between ADP and TD

---

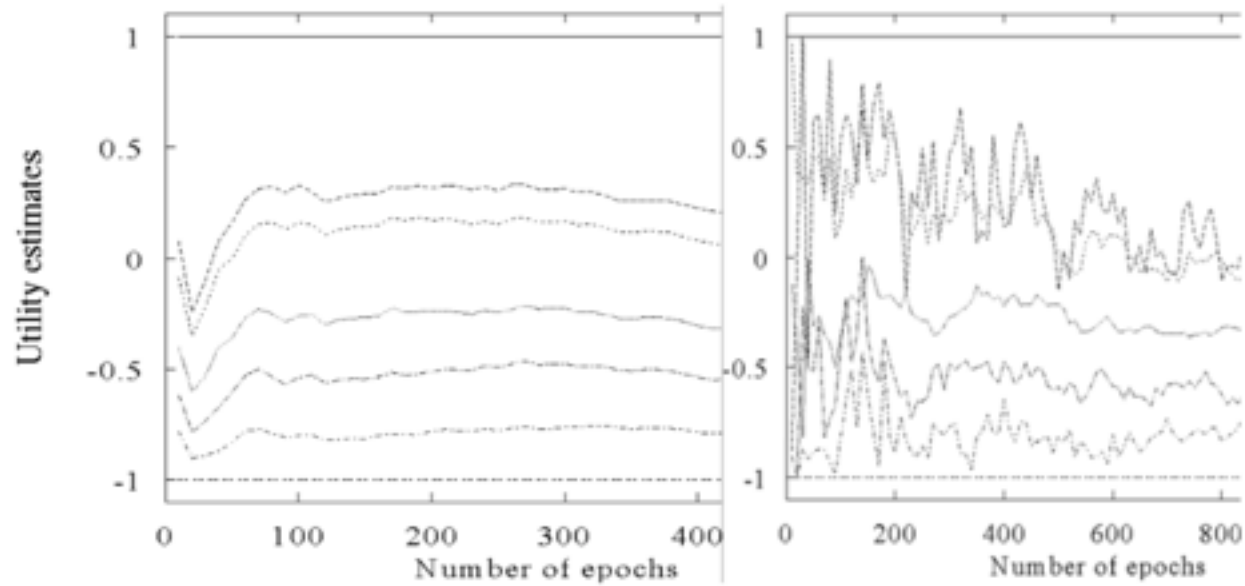
- Advantages of ADP

- Converges to true utilities faster
- Utility estimates do not vary as much from the true utilities

- Advantages of TD

- Simpler, less computation per observation
- Crude but efficient first approximation to ADP
- Do not need to build a transition model to perform its updates
  - important because we can interleave computation with exploration rather than having to wait for the whole model to be built first

# ADP and TD



# Overall comparisons

---

- DUE (model-free)
  - Simple to implement
  - Each update is fast
  - Does not exploit Bellman constraints and converges slowly
- ADP (model-based)
  - Harder to implement
  - Each update is a full policy evaluation (expensive)
  - Fully exploits Bellman constraints
  - Fast convergence (in terms of epochs)
- TD (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 DUE and ADP

# Passive Learning: Disadvantage

---

- Learning  $U^\pi(s)$  does not lead to an optimal policy, why?
- Models are incomplete/inaccurate
- Agent has only tried limited actions, we cannot gain a good overall understanding of  $P(s'|s, a)$
- This is why we need active learning



# Goal of Active Learning

---

- Let us first assume that we still have access to some sequence of trials performed by the agent
  - Agent is not following any specific policy
  - We can assume for now that the sequences should include a thorough exploration of the space
  - We will talk about how to get such sequences later
- The goal is to learn an optimal policy from such sequences
  - Active RL agents
    - Active ADP agent
    - Q-learner (based on TD algorithm)

# Active ADP Agent

---

- Model-based approach
- Using the data from its trials, agent learns a transition model  $\hat{T}$  and a reward function  $\hat{R}$
- With  $\hat{T}(s, a, s')$  and  $\hat{R}(s)$ , it has an estimate of the underlying MDP
- It can compute the optimal policy by solving the Bellman equations using value or policy iteration

$$U(s) = \hat{R}(s) + \gamma \max_a \sum_{s'} \hat{T}(s, a, s') U(s')$$

- If  $\hat{T}$  and  $\hat{R}$  are accurate estimations of the underlying MDP model, we can find the optimal policy this way

# Issues with ADP Approach

---

- Need to maintain MDP model
- $T$  can be very large,  $O(|S|^2 \cdot |A|)$
- Also, finding the optimal action requires solving the Bellman equation – time consuming
- Can we avoid this large computational complexity both in terms of time and space?

# Q-learning

---

- So far, focus on utilities for states
  - $U(s)$  = utility of state  $s$  = expected maximum future rewards
- Alternative: store Q-values
  - $Q(a, s)$  = utility of taking action  $a$  at state  $s$   
= **expected maximum future reward** if action  $a$  at state  $s$
- Relationship between  $U(s)$  and  $Q(a, s)$ ?

$$U(s) = \max_a Q(a, s)$$

# Q-learning can be model-free

---

- Note that after computing  $U(s)$ , to obtain the optimal policy, we need to compute

$$\pi(s) = \operatorname{argmax}_a \sum_{s'} T(s, a, s') U(s')$$

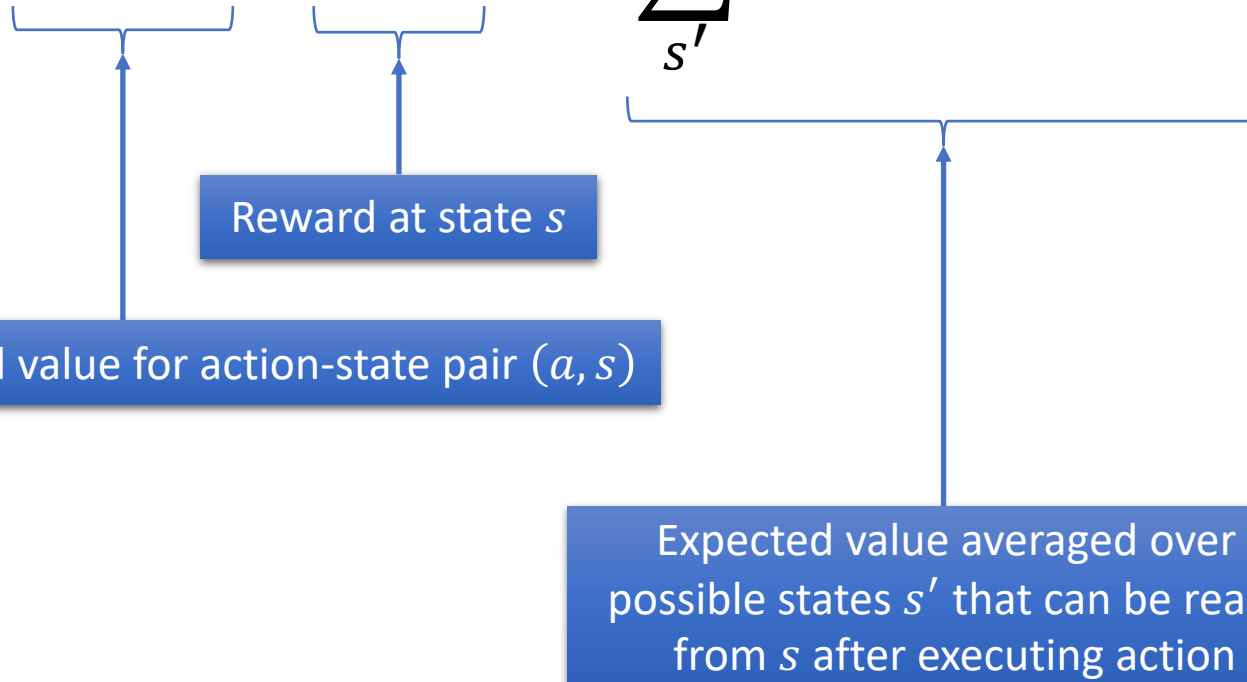
- Requires  $T$ , the model of world
- Even if we use TD learning (model-free), we still need the model to get the optimal policy
- However, if you successfully estimate  $Q(a, s)$  for all  $a$  and  $s$ , we can compute the optimal policy without using the model

$$\pi(s) = \operatorname{argmax}_a Q(a, s)$$

# Q-learning

- At equilibrium when Q-values are correct, we can write the constraint equation:

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



# Q-learning

- At equilibrium when Q-values are correct, we can write the constraint equation:

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

Expected value for action-state pair  $(a, s)$

Reward at state  $s$

Expected value averaged over all possible states  $s'$  that can be reached from  $s$  after executing action  $a$

Best value at the next state = max over all actions in state  $s'$

# Q-learning without a Model

- **Q-update:** after moving from  $s$  to state  $s'$  using action  $a$

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

Diagram illustrating the components of the Q-learning update equation:

- New estimate of  $Q(a, s)$  (points to the left  $Q(a, s)$ )
- Old estimate of  $Q(a, s)$  (points to the  $Q(a, s)$  being subtracted)
- Learning rate  $0 < \alpha < 1$  (points to  $\alpha$ )
- Difference between old estimate  $Q(a, s)$  and the new noisy sample after taking action  $a$  (points to the term in parentheses)

- TD approach
- Transition model does not appear anywhere!
- Once converged, optimal policy can be computed without transition model
  - Completely model-free learning algorithm



# Q-learning: Convergence

---

- Guaranteed to converge to true Q-values given enough exploration
- Very general procedure
  - Because it is model-free
- Converges slower than ADP agent
  - because it is completely model-free and it does not enforce consistency among values through the model

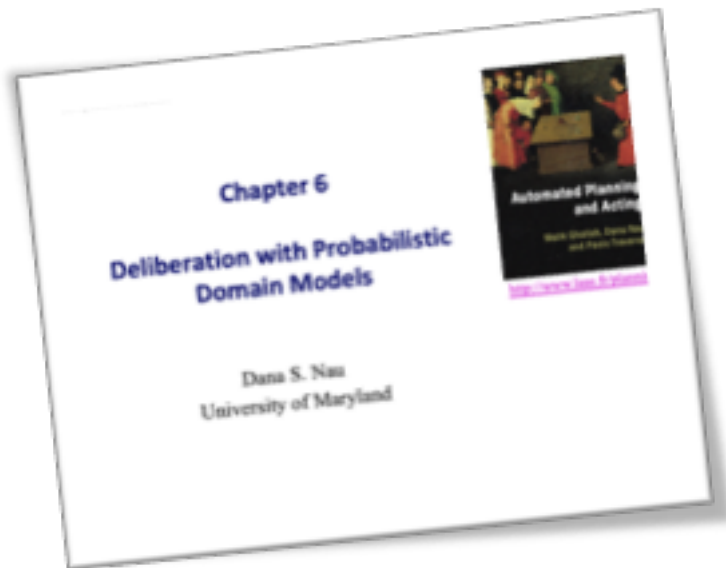
# Exploitation vs. Exploration

---

- Actions are always taken for one of the two following purposes
  - **Exploitation**: Execute the current optimal policy to get high payoff
  - **Exploration**: Try new sequences of (possibly random) actions to improve the agent's knowledge of the environment even though current model does not believe they have a high payoff
- Pure exploitation: gets stuck in a rut
- Pure exploration: not much use if you do not put that knowledge into practice

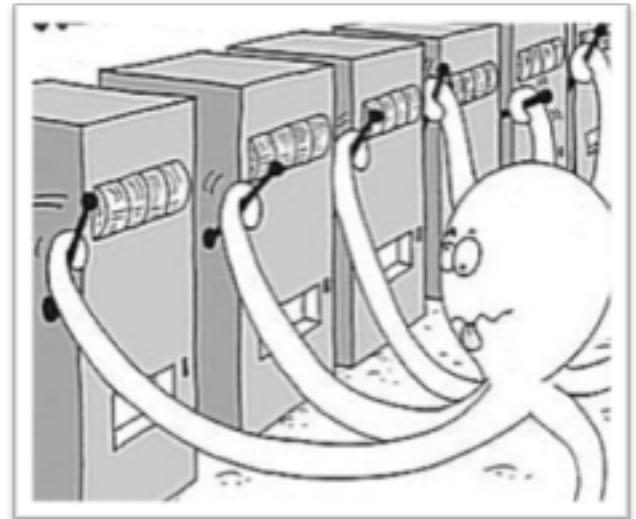
# Acknowledgements

- Slides based on material provided by Dana Nau and by Shengyu Zhang



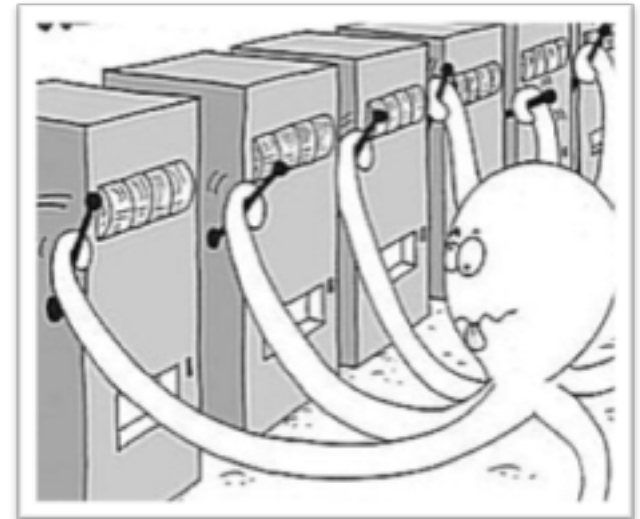
# Multi-Arm Bandit Problem

- Statistical model of sequential experiments
  - Name comes from a traditional slot machine (one-armed bandit)
- Question:  
Which machine to play?



# Actions

- $n$  arms, each with a fixed but unknown distribution of reward
  - In terms of actions: Multiple actions  $a_1, a_2, \dots, a_n$ 
    - Each  $a_i$  provides a reward from an unknown (but stationary) probability distribution  $p_i$
    - Specifically, expectation  $\mu_i$  of machine  $i$ 's reward unknown
      - If all  $\mu_i$ 's were known, then the task is easy:  
just pick  $\operatorname{argmax}_i \mu_i$
- With  $\mu_i$ 's unknown, question is which arm to pull



# Formal Model

- At each time step  $t = 1, 2, \dots, T$ :
  - Each machine  $i$  has a random reward  $X_{i,t}$ 
    - $E[X_{i,t}] = \mu_i$  independent of the past
  - Pick a machine  $I_t$  and get reward  $X_{I_t,t}$
  - Other machines' rewards hidden
- Over  $T$  time steps, we have a total reward of  $\sum_{t=1}^T X_{I_t,t}$ 
  - If all  $\mu_i$ 's known, we would have selected  $\operatorname{argmax}_i \mu_i$  at each time  $t$ 
    - Expected total reward  $T \cdot \max_i \mu_i$

- Our “regret”:  
$$T \cdot \max_i \mu_i - \sum_{t=1}^T X_{I_t,t}$$

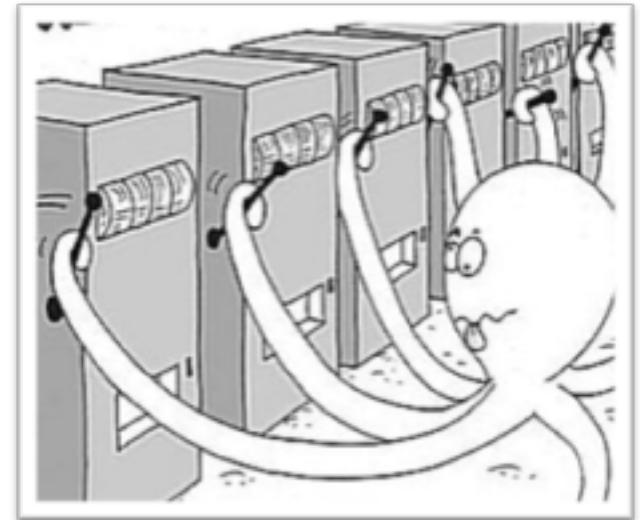
best machine's  
reward  
(in expectation)

our reward

# Exploitation vs. Exploration Dilemma

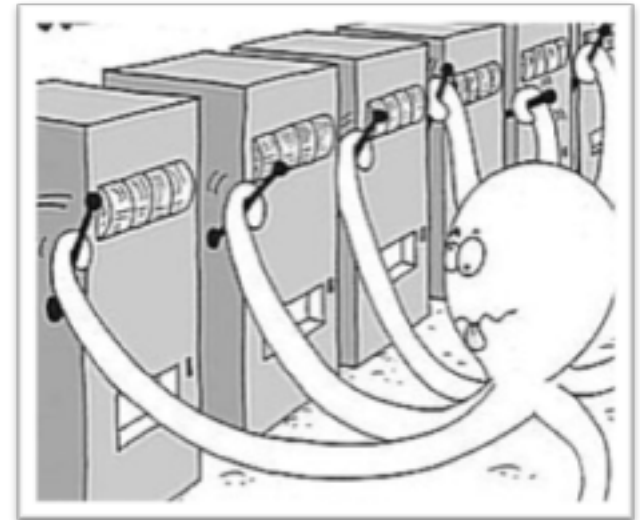
- **Exploration:** to find the best.
  - Overhead: big loss when trying the bad arms.
- **Exploitation:** to exploit what we've discovered
  - weakness: there may be better ones that we haven't explored and identified.

- **Question:**  
*With the fixed budget,  
how to balance exploration  
and exploitation such that  
the total loss (or regret)  
is small?*



# Where does the loss come from?

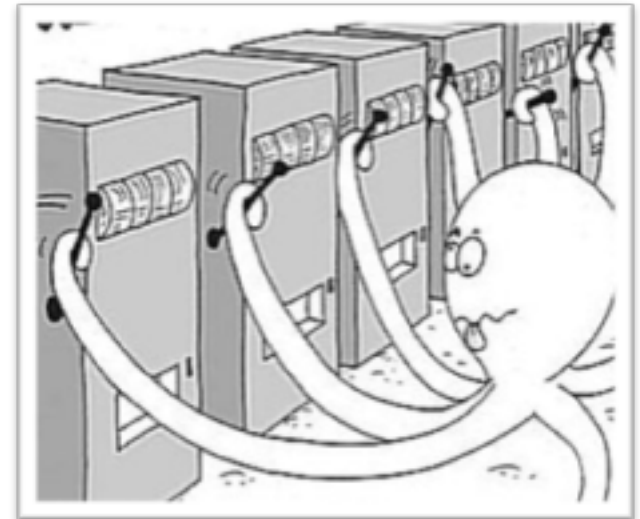
- If  $\mu_i$  is small, trying this arm too many times makes a big loss.
  - So we should try it less if we find the previous samples from it are bad
- But how to know whether an arm is good?
- The more we try an arm  $i$ , the more information we get about its distribution
  - In particular, the better estimate to its mean  $\mu_i$





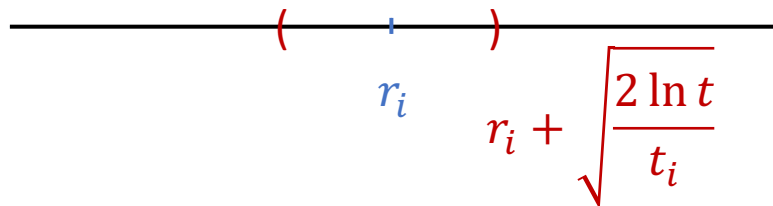
# Where does the loss come from?

- So we want to estimate each  $\mu_i$  precisely, and at the same time, we do not want to try bad arms too often
  - Two competing tasks
    - Exploration vs. exploitation dilemma
- Rough idea: we try an arm if
  - Either we have not tried it often enough
  - Or our estimate of  $\mu_i$  so far looks good



# UCB (Upper Confidence Bound) Algorithm

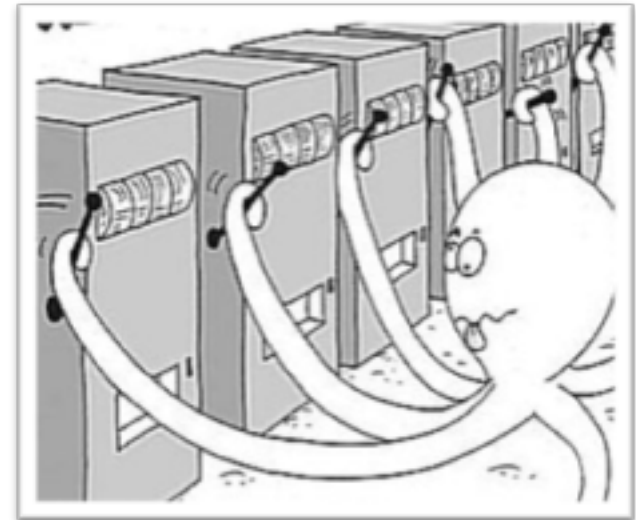
- Assume rewards between 0 and 1
  - If they are not, normalize them
- For each action  $a_i$ , let
  - $r_i$  = average reward from  $a_i$
  - $t_i$  = number of times  $a_i$  tried
  - $t = \sum_i t_i$
- Confidence interval around  $r_i$



Try each action  $a_i$  once

**loop**

choose an action  $a_i$  that has  
the highest value of  $r_i + \sqrt{2(\ln t)/t_i}$   
perform  $a_i$   
update  $r_i$ ,  $t_i$ ,  $t$

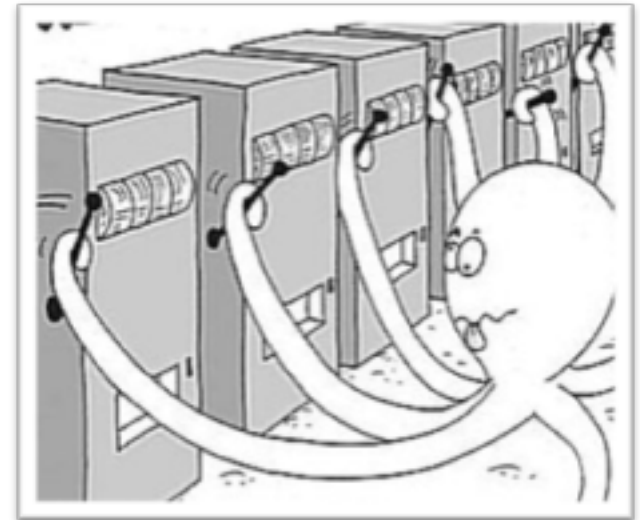


# UCB: Performance

- Theorem: If each distribution of reward has support in  $[0,1]$ , i.e., we have normalised rewards, then the regret of the UCB algorithm is at most

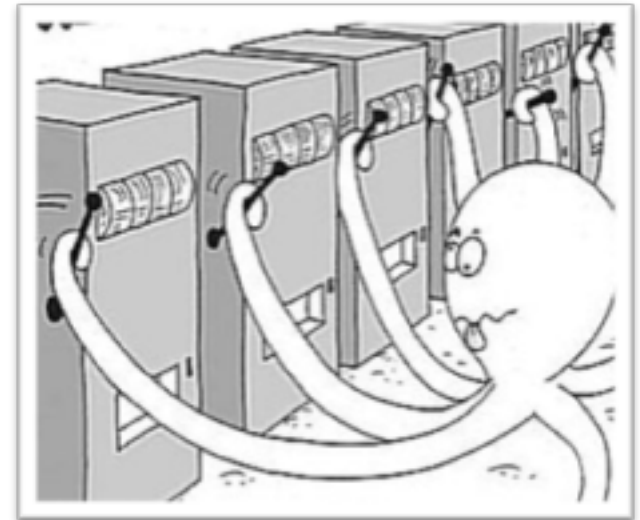
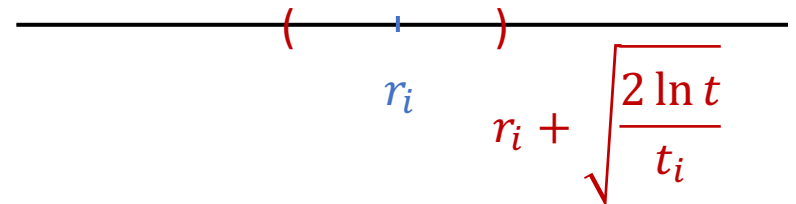
$$O\left(\sum_{i:\mu_i < \mu^*} \frac{\ln T}{\Delta_i} + \sum_{j \in \{1, \dots, n\}} \Delta_j\right)$$

- $\mu^* = \max_i \mu_i$
  - $\Delta_i = \mu^* - \mu_i$ 
    - Expected loss of choosing  $a_i$  once
  - [without proof]
- 
- Loss grows very slowly with  $T$



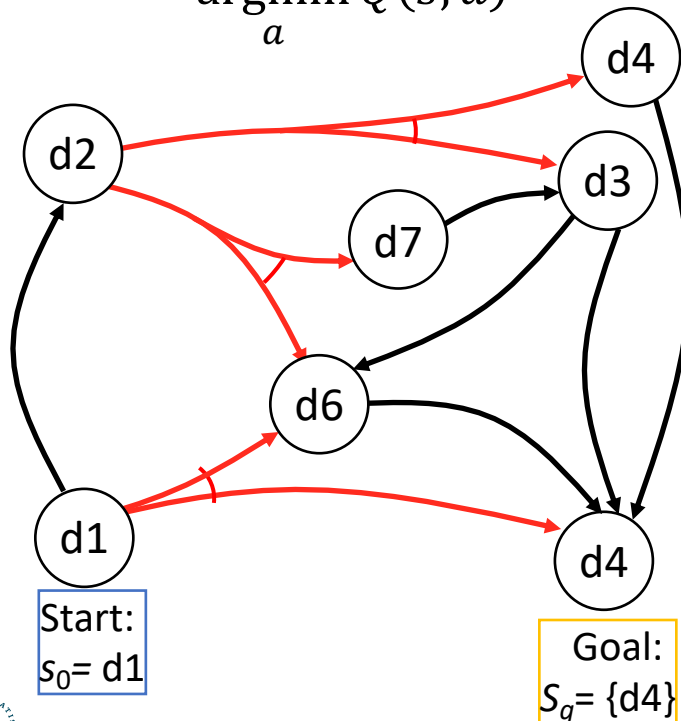
# UCB: Performance

- Uses principle of **optimism in face of uncertainty**
  - We do not have a good estimate  $\hat{\mu}_i$  of  $\mu_i$  before trying it many times
    - We thus give a big confidence interval  $[-c_i, c_i]$  for such  $i$ 
      - $c_i = \sqrt{\frac{2 \ln t}{t_i}}$
    - And select an  $i$  with maximum  $\mu_i + c_i$
  - If an action has not been tried many times, then the big confidence interval makes it still possible to be tried.
  - I.e., in face of uncertainty (of  $\mu_i$ ), we act optimistically by giving chances to those that have not been tried enough



# UCT Algorithm

- Recursive UCB computation to compute  $Q(s, a)$
- Anytime algorithm:
  - Call repeatedly until time runs out
  - Then choose action  $\operatorname{argmin}_a Q(s, a)$

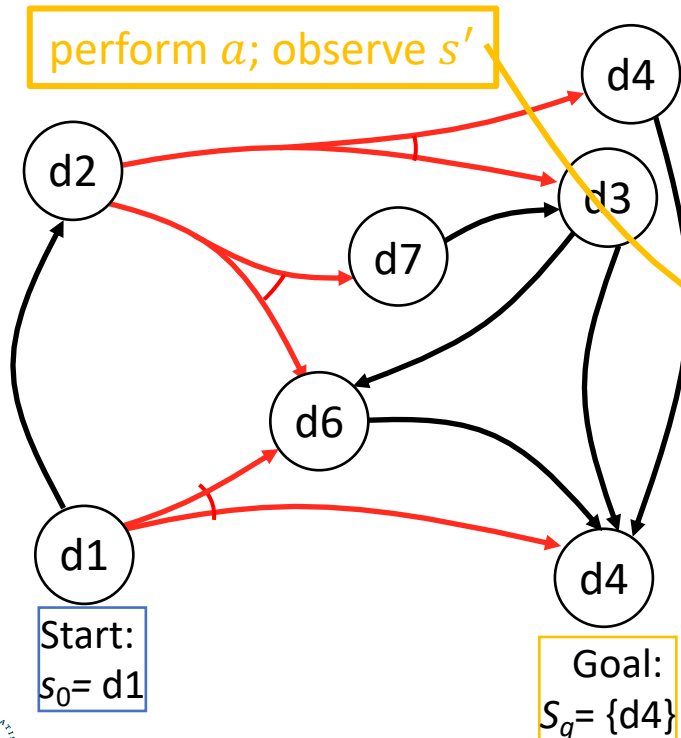


```

UCT( $s, h$ )
  if  $s \in S$  then
    return 0
  if  $h = 0$  then
    return  $V_0(s)$ 
  if  $s \notin \text{Envelope}$  then
    add  $s$  to  $\text{Envelope}$ 
     $n(s) \leftarrow 0$ 
    for all  $a \in \text{Applicable}(s)$  do
       $Q(s, a) \leftarrow 0$ 
       $n(s, a) \leftarrow 0$ 
   $\text{Untried} \leftarrow \{a \in \text{Applicable}(s) \mid n(s, a) = 0\}$ 
  if  $\text{Untried} \neq \emptyset$  then
     $\tilde{a} \leftarrow \text{Choose}(\text{Untried})$ 
  else
     $\tilde{a} \leftarrow \operatorname{argmin}_{a \in \text{Applicable}(s)} \{Q(s, a) - C \cdot [\log(n(s)) / n(s, a)]^{1/2}\}$ 
   $s' \leftarrow \text{Sample}(\Sigma, s, \tilde{a})$ 
   $\text{cost-rollout} \leftarrow \text{cost}(s, \tilde{a}) + \text{UCT}(s', h-1)$ 
   $Q(s, \tilde{a}) \leftarrow [n(s, \tilde{a}) \cdot Q(s, \tilde{a}) + \text{cost-rollout}] / (1 + n(s, \tilde{a}))$ 
   $n(s) \leftarrow n(s) + 1$ 
   $n(s, \tilde{a}) \leftarrow n(s, \tilde{a}) + 1$ 
  return  $\text{cost-rollout}$ 
    
```

# UCT as an Acting Procedure

- Suppose probabilities and costs unknown
- Suppose you can restart your actor as many times as you want
- Can modify UCT to be an acting procedure
  - Use it to explore the environment

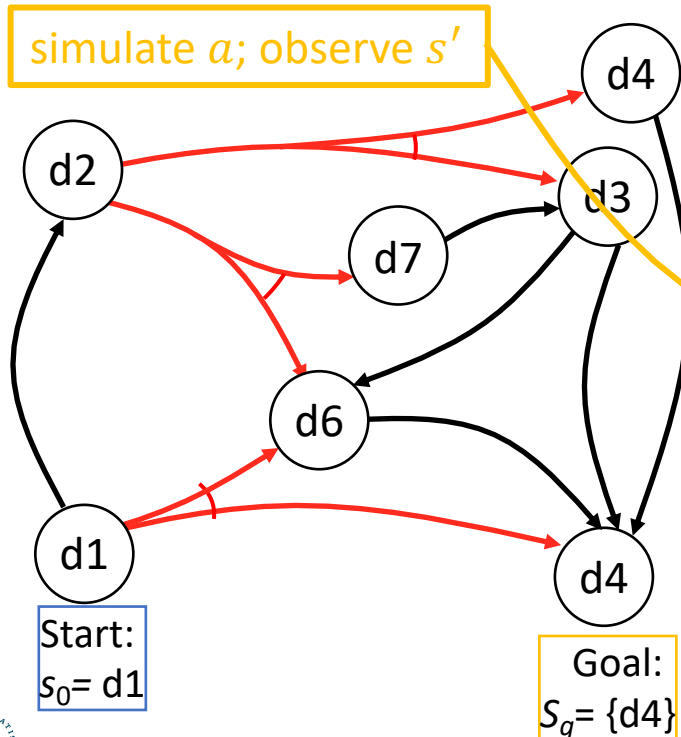


```

UCT( $s, h$ )
  if  $s \in S$  then
    return 0
  if  $h = 0$  then
    return  $V_0(s)$ 
  if  $s \notin Envelope$  then
    add  $s$  to  $Envelope$ 
     $n(s) \leftarrow 0$ 
  for all  $a \in Applicable(s)$  do
     $Q(s, a) \leftarrow 0$ 
     $n(s, a) \leftarrow 0$ 
   $Untried \leftarrow \{a \in Applicable(s) \mid n(s, a) = 0\}$ 
  if  $Untried \neq \emptyset$  then
     $\tilde{a} \leftarrow Choose(Untried)$ 
  else
     $\tilde{a} \leftarrow \operatorname{argmin}_{a \in Applicable(s)} \{Q(s, a) - C \cdot [\log(n(s)) / n(s, a)]^{1/2}\}$ 
   $s' \leftarrow Sample(\Sigma, s, \tilde{a})$ 
   $cost\_rollout \leftarrow cost(s, \tilde{a}) + UCT(s', h-1)$ 
   $Q(s, \tilde{a}) \leftarrow [n(s, \tilde{a}) \cdot Q(s, \tilde{a}) + cost\_rollout] / (1 + n(s, \tilde{a}))$ 
   $n(s) \leftarrow n(s) + 1$ 
   $n(s, \tilde{a}) \leftarrow n(s, \tilde{a}) + 1$ 
  return  $cost\_rollout$ 
    
```

# UCT as a Learning Procedure

- Suppose probabilities and costs unknown
  - But you have an accurate simulator for the environment
- Run UCT multiple times in the simulated environment
  - Learn what actions work best



```

UCT( $s, h$ )
  if  $s \in S$  then
    return 0
  if  $h = 0$  then
    return  $V_0(s)$ 
  if  $s \notin Envelope$  then
    add  $s$  to  $Envelope$ 
     $n(s) \leftarrow 0$ 
    for all  $a \in Applicable(s)$  do
       $Q(s, a) \leftarrow 0$ 
       $n(s, a) \leftarrow 0$ 
   $Untried \leftarrow \{a \in Applicable(s) \mid n(s, a) = 0\}$ 
  if  $Untried \neq \emptyset$  then
     $\tilde{a} \leftarrow Choose(Untried)$ 
  else
     $\tilde{a} \leftarrow \operatorname{argmin}_{a \in Applicable(s)} \{Q(s, a) - C \cdot [\log(n(s)) / n(s, a)]^{1/2}\}$ 
   $s' \leftarrow Sample(\Sigma, s, \tilde{a})$ 
   $cost\_rollout \leftarrow cost(s, \tilde{a}) + UCT(s', h-1)$ 
   $Q(s, \tilde{a}) \leftarrow [n(s, \tilde{a}) \cdot Q(s, \tilde{a}) + cost\_rollout] / (1 + n(s, \tilde{a}))$ 
   $n(s) \leftarrow n(s) + 1$ 
   $n(s, \tilde{a}) \leftarrow n(s, \tilde{a}) + 1$ 
  return  $cost\_rollout$ 
    
```

# UCT in Two-Player Games

- Generate Monte Carlo rollouts using a modified version of UCT
  - Rollout: game is played out to very end by selecting moves at random, result of each playout used to weight nodes in game tree
- Main differences:
  - Instead of choosing actions that minimize accumulated cost, choose actions that maximize payoff at the end of the game
  - UCT for player 1 recursively calls UCT for player 2
    - Choose opponent's action
  - UCT for player 2 recursively calls UCT for player 1
- Produced the first computer programs to play Go well
  - $\approx$  2008–2012
- Monte Carlo rollout techniques similar to UCT were used to train AlphaGo





# Intermediate Summary

---

- Passive learning
  - DUE
  - ADP
  - TD
- Active learning
  - Active ADP
  - Q-learning
- Multi-armed bandit problem
  - UCB, UCT

# Outline

---

## *Markov decision problem (MDP) recap*

- MDP formalism
- Value iteration, policy iteration

## *Partially observable Markov decision problem (POMDP)*

- POMDP agent, belief state, belief MDP
- Conditional plans, value iteration

## *Dynamic graphical models for online decision making*

- Dynamic Bayes nets
- Parameterised dynamic decision models

## *Reinforcement Learning (RL)*

- Active/passive RL
- Model-based/model-free RL
- Multi-armed bandit problem

⇒ Next: Provably Beneficial AI