AIC-4301C Artificial Intelligence
Part 2

Yasmina Abdeddaïm

e-mail: yasmina.abddeddaim@esiee.fr
Bureau: 4254
2022 - 2023

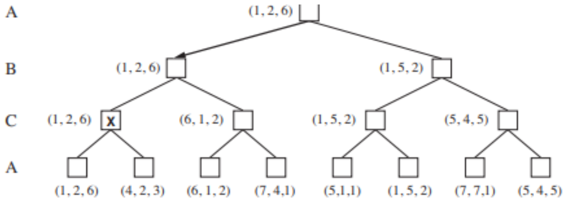
Plan

Mixed Layer Games

- Many games don't follow the exact pattern of alternation maximizer/minimizer (minimax) and maximizer/chance (expectimax) nodes
- In Pacman, after Pacman moves, there are usually multiple ghosts that take turns making moves, not a single ghost.
- Can be done by having a maximizer layer followed by consecutive ghost/minimizer layers before the second Pacman/maximizer layer.

General Games

- Different agents may have actions in a game that are not competing with another.
- Such games can be set up with trees characterized by multi-agent utilities.
- Each node is labeled with values from the viewpoint of each player.
- Multiplayer games usually involve alliances, whether formal or informal, among the players.



Plan

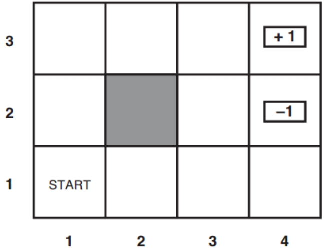
Markov Decision Processes

- We introduced the concept of **Agent**.
- We learned **traditional search problems** and how to solve them.
- We changed our model to account for **adversaries**.

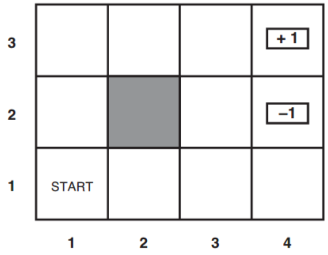
Markov Decision Processes

- We introduced the concept of **Agent**.
- We learned **traditional search problems** and how to solve them.
- We changed our model to account for **adversaries**.
- Now, we change our model to account for the **dynamic of the environment**:
  - the environment may subject the agent's actions to being nondeterministic.
  - there are multiple possible successor states that can result from an action taken in some state.
  - problems where environment have a degree of uncertainty = **nondeterministic search problems**.
  - can be solved with models known as **Markov decision processes** = **MDPs**.

Markov Decision Processes

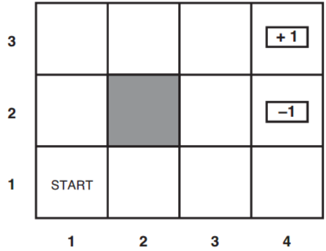


Markov Decision Processes



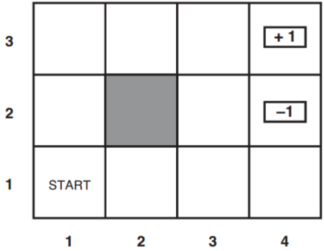
- **Initial state**: START.
- **Actions**: UP, DOWN, LEFT, RIGHT.

Markov Decision Processes



- **Initial state**: START.
- **Actions**: UP, DOWN, LEFT, RIGHT.
- **Transition system**:
  - the "intended" successor occurs with probability 0.8.
  - the agent moves at right angles to the intended direction with probability 0.2.
  - collision with a wall results in no movement.

Markov Decision Processes



- **Initial state**: START.
- **Actions**: UP, DOWN, LEFT, RIGHT.
- **Transition system**:
  - the "intended" successor occurs with probability 0.8.
  - the agent moves at right angles to the intended direction with probability 0.2.
  - collision with a wall results in no movement.
- **Goal**: two goal states have **reward** $+1$ and -1, respectively, and all other states have a reward of -0.04.

Markov Decision Processes

An MDP is defined by:

- S: a set of states
- ACTION: set of actions.
- s_0: initial state.
- T(s, a, s'): a transition function with **P(s'/ s, a)** the conditional probability that the action a from s leads to s'.
- R(s, a, s'): a **reward** function, sometimes defined only for states, R(s).
- A terminal state (not always defined).

MDP Example

A robot car wants to travel far and quickly:

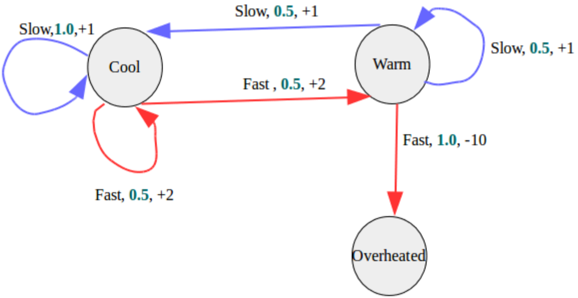
- Three states: Cool, Warm, Overheated.
- Initial state: Cool.
- Two actions: Slow, Fast.
- Going fast gets reward $+2$: $R(s, Fast, s') = +2$
- Going slow gets reward $+1$: $R(s, Slow, s') = +1$

MDP Example

A robot car wants to travel far and quickly:

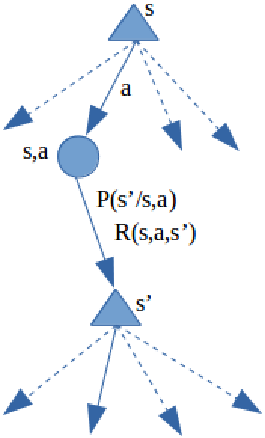
- Three states: Cool, Warm, Overheated.
- Initial state: Cool.
- Two actions: Slow, Fast.
- Going fast gets reward +2: $R(s, Fast, s') = +2$
- Going slow gets reward +1: $R(s, Slow, s') = +1$

Transition Function:

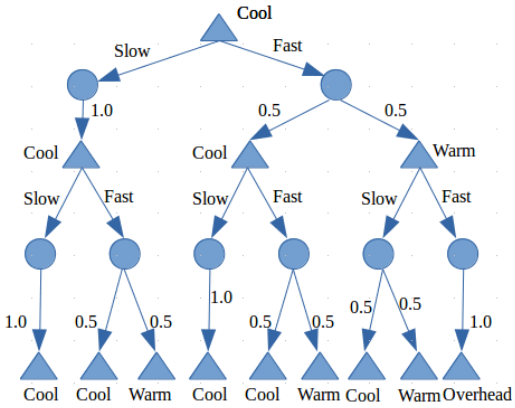


MDP Search Tree

- **s** is a state.
- **(s, a)** is a q-state = take the action a from state s.



MDP Search Tree Example



MDP Sequences

- An agent goes through different MDP states over time with discrete timesteps.
- $s_t \in S$ the state in which an agent is at timestep t.
- $a_t \in ACTION(s_t)$ the action which an agent takes at timestep t.
- s_0: initial state.
- The execution of an agent through a MDP is modeled as:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

- We call $[s_0, a_0, s_1, a_1, s_2, a_2 \dots]$ a sequence.

Markov ?

- Markovian property: the probability of reaching s' from s depends only on s and not on the history of earlier states = memoryless property.

Markov ?

- Markovian property: the probability of reaching s' from s depends only on s and not on the history of earlier states = memoryless property.
- Formally:
  - S_t denotes the random variable representing agent's state at time t.
  - A_t denotes the random variable representing the action the agent takes at time t.
  - $P(S_{t+1} = s_{t+1} / S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \ldots, S_0 = s_0)$

$$= P(S_{t+1} = s_{t+1} / S_t = s_t, A_t = a_t)$$

Markov ?

- Markovian property: the probability of reaching s' from s depends only on s and not on the history of earlier states = memoryless property.
- Formally:
  - S_t denotes the random variable representing agent's state at time t.
  - A_t denotes the random variable representing the action the agent takes at time t.
  - $P(S_{t+1} = s_{t+1} / S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \ldots, S_0 = s_0)$
$$= P(S_{t+1} = s_{t+1} / S_t = s_t, A_t = a_t)$$
- T(s, a, s') = P(s'/ s, a).

Policy

- A solution of a deterministic agent search problem is a sequence of actions from start to a goal.
- An optimal solution for a search problem returns the sequence with the smallest total cost.

Policy

- A solution of a deterministic agent search problem is a sequence of actions from start to a goal.
- An optimal solution for a search problem returns the sequence with the smallest total cost.
- For MDPs a solution is a policy $\pi : S \rightarrow ACTION$ i.e, a function that returns an action for each state.

Policy

- A solution of a deterministic agent search problem is a sequence of actions from start to a goal.
- An optimal solution for a search problem returns the sequence with the smallest total cost.
- For MDPs a solution is a policy $\pi : S \rightarrow ACTION$ i.e, a function that returns an action for each state.
- An optimal policy π^* maximizes **expected utility** with $U([s_0, a_0, s_1, \ldots, s_n])$ is the utility of sequence $[s_0, a_0, s_1, \ldots, s_n]$.

Policy

- A solution of a deterministic agent search problem is a sequence of actions from start to a goal.
- An optimal solution for a search problem returns the sequence with the smallest total cost.
- For MDPs a solution is a policy $\pi : S \rightarrow ACTION$ i.e, a function that returns an action for each state.
- An optimal policy π^* maximizes **expected utility** with $U([s_0, a_0, s_1, \ldots, s_n])$ is the utility of sequence $[s_0, a_0, s_1, \ldots, s_n]$.
- A policy function defines a reflex agent.

Policy

- A solution of a deterministic agent search problem is a sequence of actions from start to a goal.
- An optimal solution for a search problem returns the sequence with the smallest total cost.
- For MDPs a solution is a policy $\pi : S \rightarrow ACTION$ i.e, a function that returns an action for each state.
- An optimal policy π^* maximizes **expected utility** with $U([s_0, a_0, s_1, \ldots, s_n])$ is the utility of sequence $[s_0, a_0, s_1, \ldots, s_n]$.
- A policy function defines a reflex agent.
- Expectimax didn't compute entire policies, it computes the action to take for a single state only.

Finite horizon/ Infinite horizon

A finite horizon decision means that there is a fixed timestep N after which nothing matters.

Finite horizon/ Infinite horizon

A finite horizon decision means that there is a fixed timestep N after which nothing matters.

$$\forall k > 0, U([s_0, a_0, s_1, a_1, \ldots, s_{N+k}]) = U([s_0, a_0, s_1, a_1, \ldots, s_N])$$

Stationary

- If tomorrow you prefer one future to another, then you should still prefer that future today.

Stationary

- If tomorrow you prefer one future to another, then you should still prefer that future today.
- If two state sequences $[s_0, a_0, s_1, a_1, \ldots, s_n]$ and $[s'_0, a'_0, s'_1, a'_1, \ldots, s'_n]$ begin with the same state (i.e., $s_0 = s'_0$), then the two sequences should be preference-ordered the same way as the sequences $[s_1, a_1, \ldots, s_n]$ and $[s'_1, a'_1, \ldots, s'_n]$.

Stationary

- If tomorrow you prefer one future to another, then you should still prefer that future today.
- If two state sequences $[s_0, a_0, s_1, a_1, \ldots, s_n]$ and $[s'_0, a'_0, s'_1, a'_1, \ldots, s'_n]$ begin with the same state (i.e., $s_0 = s'_0$), then the two sequences should be preference-ordered the same way as the sequences $[s_1, a_1, \ldots, s_n]$ and $[s'_1, a'_1, \ldots, s'_n]$.
- Under stationarity there are just two ways to assign utilities to sequences: Additive utility and Discounted utility.

Additive utility

$$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + \ldots$$

Discounted utility

$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$
where $|\gamma| \leq 1$ is the discount factor.

Discounted utility

$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$
where $|\gamma| \leq 1$ is the discount factor.

- The discount factor describes the preference of an agent for current rewards over future rewards.

Discounted utility

$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$
where $|\gamma| \leq 1$ is the discount factor.

- The discount factor describes the preference of an agent for current rewards over future rewards.
- When γ is close to 0, rewards in the distant future are viewed as insignificant.

Discounted utility

$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$
where $|\gamma| \leq 1$ is the discount factor.

- The discount factor describes the preference of an agent for current rewards over future rewards.
- When γ is close to 0, rewards in the distant future are viewed as insignificant.
- When γ is 1, discounted utility are exactly equivalent to additive utility.

Discounted utility

$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$
where $|\gamma| \leq 1$ is the discount factor.

- The discount factor describes the preference of an agent for current rewards over future rewards.
- When γ is close to 0, rewards in the distant future are viewed as insignificant.
- When γ is 1, discounted utility are exactly equivalent to additive utility.
- Values in the range $-1 \leq \gamma \leq 0$ are not meaningful in most real-world situations, a negative value for γ means the reward for a state s would flip-flop between positive and negative values at alternating timesteps.

Discounted utility

$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$
where $|\gamma| \leq 1$ is the discount factor.

- The discount factor describes the preference of an agent for current rewards over future rewards.
- When γ is close to 0, rewards in the distant future are viewed as insignificant.
- When γ is 1, discounted utility are exactly equivalent to additive utility.
- Values in the range $-1 \leq \gamma \leq 0$ are not meaningful in most real-world situations, a negative value for γ means the reward for a state s would flip-flop between positive and negative values at alternating timesteps.
- Discounting appears to be a good model of human preferences over time.

Discounted utility

$$U([s_0, a_0, s_1, \ldots]) \leq R_{max}/(1-\gamma) \text{ } if \text{ } \gamma < 1$$

where R_{max} is the maximum possible reward reachable at any given timestep in the MDP.

Discounted utility

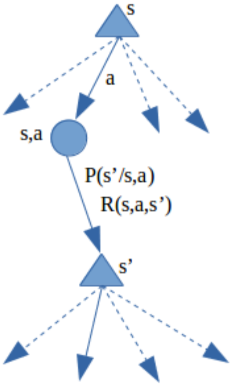
$$U([s_0, a_0, s_1, \ldots]) \leq R_{max}/(1-\gamma) \; if \; \gamma < 1$$

where R_{max} is the maximum possible reward reachable at any given timestep in the MDP.

$$\begin{array}{rcl} U([s_0, a_0, s_1, \ldots]) & = & R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots \\ & = & \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \\ & \leq & \sum_{t=0}^{\infty} \gamma^t R_{max} \\ & \leq & R_{max}(1-\gamma^{\infty})/(1-\gamma) \\ & \leq & R_{max}/(1-\gamma) \; if \; \gamma < 1 \end{array}$$

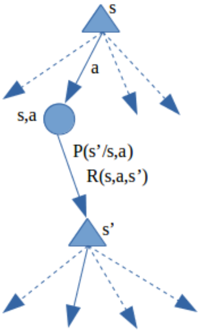
Optimality

- $V^*(s)$: expected utility starting in s and acting optimally.
- $Q^*(s,a)$: expected utility having taken action a from state s and (thereafter) acting optimally.
- $\pi^*(s)$: optimal action from state s.



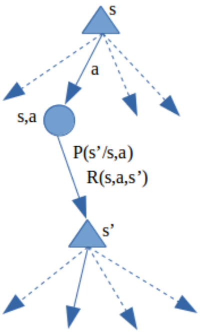
Optimality Recursive Equations

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



Optimality Recursive Equations

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



The Bellman Equation:

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

Value Iteration (An algorithm to solve MDPs)

A dynamic programming algorithm that uses an iteratively longer time limit to compute time-limited values until convergence ($\forall s, V_{k+1}(s) = V_k(s)$).

1. $\forall s \in S, V_0(s) = 0$
2. Repeat the following update rule until convergence:
   $\forall s \in S, V_{k+1}(s) = max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$

Value Iteration Example

$V_1(cool) = max\{1 \times [1 + 0.5 \times 0], 0.5 \times [2 + 0.5 \times 0] + 0.5 \times [2 + 0.5 \times 0]\} =$
$max\{1, 2\} = 2$
$V_1(warm) = max\{0.5 \times [1 + 0.5 \times 0] + 0.5 \times [1 + 0.5 \times 0], 1 \times [-10 + 0.5 \times 0]\} =$
$max\{1, -10\} = 1$
$V_1(overheated) = max\{\} = 0$

Value Iteration Example

$V_1(cool) = max\{1 \times [1 + 0.5 \times 0], 0.5 \times [2 + 0.5 \times 0] + 0.5 \times [2 + 0.5 \times 0]\} =$
$max\{1, 2\} = 2$
$V_1(warm) = max\{0.5 \times [1 + 0.5 \times 0] + 0.5 \times [1 + 0.5 \times 0], 1 \times [-10 + 0.5 \times 0]\} =$
$max\{1, -10\} = 1$
$V_1(overheated) = max\{\} = 0$

$V_2(cool) = max\{1 \times [1 + 0.5 \times 2], 0.5 \times [2 + 0.5 \times 2] + 0.5 \times [2 + 0.5 \times 1]\} =$
$max\{2, 2.75\} = 2.75$
$V_2(warm) = max\{0.5 \times [1 + 0.5 \times 2] + 0.5 \times [1 + 0.5 \times 1], 1 \times [-10 + 0.5 \times 0]\} =$
$max\{1.75, -10\} = 1.75$
$V_2(overheated) = max\{\} = 0$

Policy Extraction

- In every state take the action a which yields the maximum expected utility.
- a is the action which takes us to the q-state with maximum q-value: $\forall s \in S$,

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

we must recompute all necessary q-values with the Bellman equation before applying argmax.

Complexity

- At each iteration, we must update the values of all $|S|$ states, each of which requires iteration over all $|A|$ actions as we compute the q-value for each action.
- The computation of each of these q-values, in turn, requires iteration over each of the $|S|$ states again
- Time complexity $= O(|S|^2|A|)$.

Policy Iteration

An algorithm that maintains the optimality of value iteration while providing significant performance gains.

Policy Iteration

An algorithm that maintains the optimality of value iteration while providing significant performance gains.

- Policy evaluation: calculate utilities for some fixed policy (not optimal utilities).

Policy Iteration

An algorithm that maintains the optimality of value iteration while providing significant performance gains.

- Policy evaluation: calculate utilities for some fixed policy (not optimal utilities).
- Policy improvement: update policy using one-step look-ahead with resulting policy evaluation.

Policy Iteration

An algorithm that maintains the optimality of value iteration while providing significant performance gains.

- Policy evaluation: calculate utilities for some fixed policy (not optimal utilities).
- Policy improvement: update policy using one-step look-ahead with resulting policy evaluation.
- Repeat steps until policy converges.

Policy Iteration Algorithm

1. Define an initial policy π_0.
   This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.

Policy Iteration Algorithm

1. Define an initial policy π_0.
   This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until $\pi_{i+1} = \pi_i$, if $\pi_{i+1} = \pi_i$ then $\pi^* = \pi_i$:

Policy Iteration Algorithm

1. Define an initial policy π_0.
   This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until $\pi_{i+1} = \pi_i$, if $\pi_{i+1} = \pi_i$ then $\pi^* = \pi_i$:
   1. Evaluate the current policy π_i with policy evaluation.

$$V^{\pi_i}(s) = \cancel{max_a} \sum_{s'} T(s, \cancel{a}\ \pi_i(s), s')[R(s, \cancel{a}\ \pi_i(s), s') + \gamma V^{\pi_i}(s')$$

$V^{\pi_i}(s)$ can be computed by:

- Method 1: solving the generated system of $|S|$ equations system.
- Method 2: using the following update rule until convergence
  $V^{\pi_i}_{k+1}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V^{\pi_i}_k(s')]$
  just like in value iteration, this method is typically slower in practice

Policy Iteration Algorithm

1. Define an initial policy π_0.
   This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.
2. Repeat the following until $\pi_{i+1} = \pi_i$, if $\pi_{i+1} = \pi_i$ then $\pi^* = \pi_i$:
   1. Evaluate the current policy π_i with policy evaluation.

      $$V^{\pi_i}(s) = \cancel{max_a} \sum_{s'} T(s, \cancel{a}\ \pi_i(s), s')[R(s, \cancel{a}\ \pi_i(s), s') + \gamma V^{\pi_i}(s')$$

      $V^{\pi_i}(s)$ can be computed by:
      - Method 1: solving the generated system of $|S|$ equations system.
      - Method 2: using the following update rule until convergence
        $V^{\pi_i}_{k+1}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V^{\pi_i}_k(s')]$
        just like in value iteration, this method is typically slower in practice
   2. Policy improvement of π_i to generate a better policy π_{i+1}.
      Uses policy extraction on the values of states generated by policy evaluation.

      $$\pi_{i+1}(s) = argmax_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')]$$

Policy Iteration Example

1. Initial policy π_0:

$$\pi_0(cool) = slow, \pi_0(warm) = slow, \pi_0(overheated) = -$$

Policy Iteration Example

1. Initial policy π_0:

$$\pi_0(cool) = slow, \pi_0(warm) = slow, \pi_0(overheated) = -$$

2. Policy evaluation π_0:

$$\begin{cases} V^{\pi_0}(cool) & = 1 \times [1 + 0.5 \times V^{\pi_0}(cool)] \\ V^{\pi_0}(warm) & = 0.5 \times [1 + 0.5 \times V^{\pi_0}(cool)] + \\ & \quad 0.5 \times [1 + 0.5 \times V^{\pi_0}(warm)] \end{cases}$$

Policy Iteration Example

1. Initial policy π_0:

$$\pi_0(cool) = slow, \pi_0(warm) = slow, \pi_0(overheated) = -$$

2. Policy evaluation π_0:

$$\begin{cases} V^{\pi_0}(cool) & = 1 \times [1 + 0.5 \times V^{\pi_0}(cool)] \\ V^{\pi_0}(warm) & = 0.5 \times [1 + 0.5 \times V^{\pi_0}(cool)] + \\ & 0.5 \times [1 + 0.5 \times V^{\pi_0}(warm)] \end{cases}$$

$$V^{\pi_0}(cool) = 2,\ V^{\pi_0}(warm) = 2,\ V^{\pi_0}(overheated) = 0$$

Policy Iteration Example

1. Initial policy π_0:

$$\pi_0(cool) = slow, \pi_0(warm) = slow, \pi_0(overheated) = -$$

2. Policy evaluation π_0:

$$\begin{cases} V^{\pi_0}(cool) & = 1 \times [1 + 0.5 \times V^{\pi_0}(cool)] \\ V^{\pi_0}(warm) & = 0.5 \times [1 + 0.5 \times V^{\pi_0}(cool)] + \\ & 0.5 \times [1 + 0.5 \times V^{\pi_0}(warm)] \end{cases}$$

$$V^{\pi_0}(cool) = 2,\ V^{\pi_0}(warm) = 2,\ V^{\pi_0}(overheated) = 0$$

3. Policy improvement of π_0:

$$\begin{cases} \pi_1(cool) & = argmax\{slow : 1 \times [1 + 0.5 \times 2], \\ & fast : 0.5 \times [2 + 0.5 \times 2] + 0.5 \times [2 + 0.5 \times 2]\} = fast \\ \pi_1(warm) & = argmax\{slow : 0.5 \times [1 + 0.5 \times 2] + 0.5 \times [1 + 0.5 \times 2], \\ & fast : 1 \times [-10 + 0.5 \times 0]\} = slow \end{cases}$$

Policy Iteration Example

1. Initial policy π_0:

$$\pi_0(cool) = slow, \pi_0(warm) = slow, \pi_0(overheated) = -$$

2. Policy evaluation π_0:

$$\begin{cases} V^{\pi_0}(cool) & = 1 \times [1 + 0.5 \times V^{\pi_0}(cool)] \\ V^{\pi_0}(warm) & = 0.5 \times [1 + 0.5 \times V^{\pi_0}(cool)] + \\ & 0.5 \times [1 + 0.5 \times V^{\pi_0}(warm)] \end{cases}$$

$$V^{\pi_0}(cool) = 2, V^{\pi_0}(warm) = 2, V^{\pi_0}(overheated) = 0$$

3. Policy improvement of π_0:

$$\begin{cases} \pi_1(cool) & = argmax\{slow : 1 \times [1 + 0.5 \times 2], \\ & fast : 0.5 \times [2 + 0.5 \times 2] + 0.5 \times [2 + 0.5 \times 2]\} = fast \\ \pi_1(warm) & = argmax\{slow : 0.5 \times [1 + 0.5 \times 2] + 0.5 \times [1 + 0.5 \times 2], \\ & fast : 1 \times [-10 + 0.5 \times 0]\} = slow \end{cases}$$

$$\pi_1(cool) = fast, \pi_1(warm) = slow, \pi_1(overheated) = -$$

Policy Iteration Example

1. Initial policy π_0:

$$\pi_0(cool) = slow, \pi_0(warm) = slow, \pi_0(overheated) = -$$

2. Policy evaluation π_0:

$$\begin{cases} V^{\pi_0}(cool) & = 1 \times [1 + 0.5 \times V^{\pi_0}(cool)] \\ V^{\pi_0}(warm) & = 0.5 \times [1 + 0.5 \times V^{\pi_0}(cool)] + \\ & 0.5 \times [1 + 0.5 \times V^{\pi_0}(warm)] \end{cases}$$

$$V^{\pi_0}(cool) = 2, V^{\pi_0}(warm) = 2, V^{\pi_0}(overheated) = 0$$

3. Policy improvement of π_0:

$$\begin{cases} \pi_1(cool) & = argmax\{slow : 1 \times [1 + 0.5 \times 2], \\ & fast : 0.5 \times [2 + 0.5 \times 2] + 0.5 \times [2 + 0.5 \times 2]\} = fast \\ \pi_1(warm) & = argmax\{slow : 0.5 \times [1 + 0.5 \times 2] + 0.5 \times [1 + 0.5 \times 2], \\ & fast : 1 \times [-10 + 0.5 \times 0]\} = slow \end{cases}$$

$$\pi_1(cool) = fast, \pi_1(warm) = slow, \pi_1(overheated) = -$$

4. Policy evaluation π_1:

...

Plan

1. Adversarial Search
2. Markov Decision Processes
3. Reinforcement Learning

Machine Learning

- Supervised Learning:
  - Learning with labeled instances
  - Ex. : Decision Trees, Neural Networks, SVMS, ...
- Unsupervised Learning:
  - Learning without labels
  - Ex. : K-means, clustering, ...

Machine Learning

- Supervised Learning:
  - Learning with labeled instances
  - Ex. : Decision Trees, Neural Networks, SVMS, ...
- Unsupervised Learning:
  - Learning without labels
  - Ex. : K-means, clustering, ...
- Reinforcement Learning:
  - Learning with rewards
  - App. : robots, autonomous vehicles, ...

Rewards

- Rewards were introduced in MDPs (Markov Decisions processes).
- An optimal policy is a policy that maximizes the expected total utility (utilities are computed using rewards).

Rewards

- Rewards were introduced in MDPs (Markov Decisions processes).
- An optimal policy is a policy that maximizes the expected total utility (utilities are computed using rewards).
- Reinforcement learning uses the observed rewards to learn an optimal (or nearly optimal) policy.

Rewards

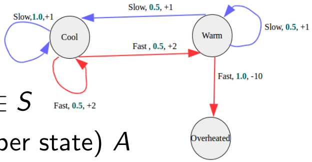
- Rewards were introduced in MDPs (Markov Decisions processes).
- An optimal policy is a policy that maximizes the expected total utility (utilities are computed using rewards).
- Reinforcement learning uses the observed rewards to learn an optimal (or nearly optimal) policy.
- In MDPs the agent has a complete model of the environment and knows the reward function.

Rewards

- Rewards were introduced in MDPs (Markov Decisions processes).
- An optimal policy is a policy that maximizes the expected total utility (utilities are computed using rewards).
- Reinforcement learning uses the observed rewards to learn an optimal (or nearly optimal) policy.
- In MDPs the agent has a complete model of the environment and knows the reward function.
- In reinforcement learning we assume no knowledge of the environment and the reward function.

Reinforcement Learning

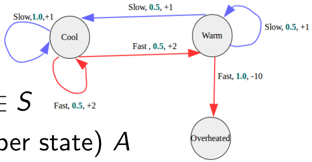
A Markov decision process (MDP):



- A set of states $s \in S$
- A set of actions (per state) A
- A model $T(s, a, s')$
- A reward function $R(s, a, s')$
- Looking for a policy $\pi(s)$

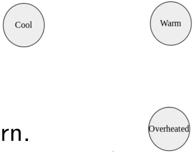
Reinforcement Learning

A Markov decision process (MDP):



- A set of states $s \in S$
- A set of actions (per state) A
- A model $T(s, a, s')$
- A reward function $R(s, a, s')$
- Looking for a policy $\pi(s)$

Reinforcement Learning:



- A MDP.
- Don't know T or R.
- Must try actions to learn.

Exploration

- Exploration step: At each time step an agent starts in a state s, then takes an action a and ends up in a successor state s' reaching some reward r until arriving at a terminal state.

Exploration

- Exploration step: At each time step an agent starts in a state s, then takes an action a and ends up in a successor state s' reaching some reward r until arriving at a terminal state.
- A sample is a tuple (s, a, s', r).

Exploration

- Exploration step: At each time step an agent starts in a state s, then takes an action a and ends up in a successor state s' reaching some reward r until arriving at a terminal state.
- A sample is a tuple (s, a, s', r).
- An episode is a collection of samples.

Exploration

- Exploration step: At each time step an agent starts in a state s, then takes an action a and ends up in a successor state s' reaching some reward r until arriving at a terminal state.
- A sample is a tuple (s, a, s', r).
- An episode is a collection of samples.

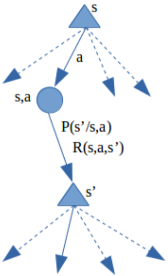
Agents go through many episodes during exploration in order to collect sufficient data needed for learning.

Model-based/ Model-free Learning

- Model-based learning: estimate the transition and reward functions with the samples reached during exploration before using these estimates to solve the MDP normally with value or policy iteration.
- Model-free learning: estimate the utilities ($V(s)$) or q-utilities ($Q(s, a)$) without ever constructing a model of the rewards and transitions in the MDP.

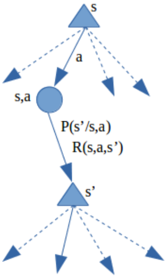
Model-based Learning

Agent generates an approximation $\hat{T}(s, a, s')$ of $T(s, a, s')$, by counting during the exploration the number of times it arrives in each state s' after entering each q-state (s, a).



Model-based Learning

Agent generates an approximation $\hat{T}(s, a, s')$ of $T(s, a, s')$, by counting during the exploration the number of times it arrives in each state s' after entering each q-state (s, a).

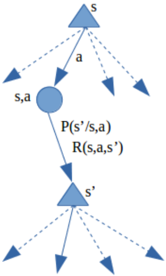


Step 1: Learn empirical MDP model

- Exploration
- Count outcomes s' for each s, a
- Normalize to give an estimate of $\hat{T}(s, a, s')$
- Discover each $\hat{R}(s, a, s')$ when we explore (s, a, s')

Model-based Learning

Agent generates an approximation $\hat{T}(s, a, s')$ of $T(s, a, s')$, by counting during the exploration the number of times it arrives in each state s' after entering each q-state (s, a).

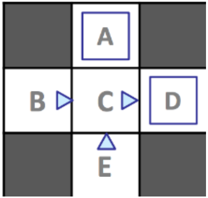


Step 1: Learn empirical MDP model

- Exploration
- Count outcomes s' for each s, a
- Normalize to give an estimate of $\hat{T}(s, a, s')$
- Discover each $\hat{R}(s, a, s')$ when we explore (s, a, s')

Step 2: Solve the learned MDP

Model-based Learning Example (with $\gamma = 1$)



Exploration	Learned MDP
Episode 1	$\hat{T}(s, a, s')$
B, east, C, -1	$\hat{T}(B, east, C) = 1.00$
C, east, D, -1	$\hat{T}(C, east, D) = 0.75$
D, exit, x, +10	$\hat{T}(C, east, A) = 0.25$
Episode 2	...
B, east, C, -1	$\hat{R}(s, a, s')$
C, east, D, -1	$\hat{R}(B, east, C) = -1$
D, exit, x, +10	$\hat{R}(C, east, D) = -1$
Episode 3	$\hat{R}(D, exit, x) = +10$
E, north, C, -1	...
C, east, D, -1	
D, exit, x, +10	
Episode 4	
E, north, C, -1	
C, east, A, -1	
A, exit, x, -10	

Model-free Learning

- Passive Reinforcement Learning: the agent is given a policy to follow and learns the utilities of states under that policy during exploration.
- Active Reinforcement Learning: the agent must also learn the policy.

Passive Reinforcement Learning

- Input: a fixed policy $\pi(s)$.
- You don't know the transitions $T(s, a, s')$.
- You don't know the rewards $R(s, a, s')$.
- Goal: learn the state utilities.

In this case:

- No choice about what actions to take (given policy).
- Just execute the policy and learn from experience.

Passive Reinforcement Learning

1. Direct Evaluation
2. Temporal Difference Learning

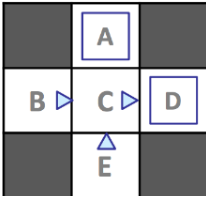
Direct Evaluation

Idea: Average observed sample utilities

- Act according to π
- Every time the agent visits a state, write down the sum of discounted utilities.
- Average those samples.

Direct Evaluation Example (with $\gamma = 1$)

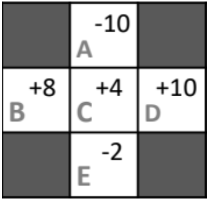
Input Policy π



Exploration
Episode 1
B, east, C, -1
C, east, D, -1
D, exit, x, +10
Episode 2
B, east, C, -1
C, east, D, -1
D, exit, x, +10
Episode 3
E, north, C, -1
C, east, D, -1
D, exit, x, +10
Episode 4
E, north, C, -1
C, east, A, -1
A, exit, x, -10

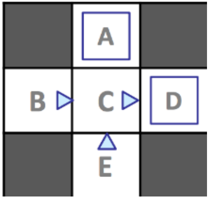
Output state utilities $V(s)$

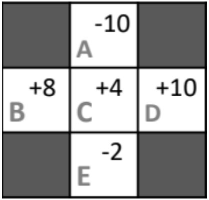
s	Total Utilities	Times Visited	$V^\pi(s)$
A	-10	1	-10
B	16	2	8
C	16	4	4
D	30	3	10
E	-4	2	-2



Direct Evaluation Problems

- It wastes information about state connections
- Each state must be learned separately, so it takes a long time to learn





If B and E both go to C under this policy, how can their values be different?

Temporal Difference Learning (TD learning)

Idea: learning from every experience rather than simply keeping track of total rewards and number of times states are visited and learning at the end as direct evaluation does.

Temporal Difference Learning (TD learning)

Idea: learning from every experience rather than simply keeping track of total rewards and number of times states are visited and learning at the end as direct evaluation does.

Reminder: policy evaluation uses the system of equations generated by a fixed policy π and the Bellman equation to determine the utilities of states under that policy.

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

Temporal Difference Learning (TD learning)

Idea: learning from every experience rather than simply keeping track of total rewards and number of times states are visited and learning at the end as direct evaluation does.

- TD learning tries to answer the question of how to compute $V^{\pi}(s)$ without the weights

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

Temporal Difference Learning (TD learning)

Idea: learning from every experience rather than simply keeping track of total rewards and number of times states are visited and learning at the end as direct evaluation does.

- TD learning tries to answer the question of how to compute $V^\pi(s)$ without the weights

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

- Uses exponential moving average: $\bar{x}_n = (\alpha - 1)\bar{x}_{n-1} + \alpha x_n$
  - $\alpha, 0 \leq \alpha \leq 1$ is a parameter known as the learning rate.
  - Makes recent samples more important.
  - Forgets about the past (distant past values were wrong anyway).

Temporal Difference Learning (TD learning)

Update $V^{\pi}(s)$ each time we experience a transition (s, a, s', r) during exploration using exponential moving.

Temporal Difference Learning (TD learning)

Update $V^{\pi}(s)$ each time we experience a transition (s, a, s', r) during exploration using exponential moving.

- compute a sample using (s, a, s', r)

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

Temporal Difference Learning (TD learning)

Update $V^\pi(s)$ each time we experience a transition (s, a, s', r) during exploration using exponential moving.

- compute a sample using (s, a, s', r)

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

- Update $V^\pi(s)$ using exponential moving average:
$V^\pi(s) = (1 - \alpha)V^\pi(s) + \alpha \ \textit{sample}$

Temporal Difference Learning Algorithm

1. Start $V^\pi(s) = 0, \forall s \in S$
2. Compute a sample:

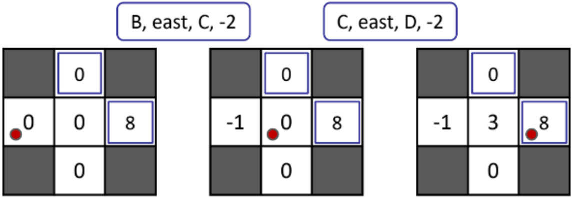
$$sample = R(s, \pi(s), s') + \gamma V^\pi(s')$$

3. Update $V^\pi(s)$: $V^\pi(s) = (1-\alpha)V^\pi(s) + \alpha\ sample$

It is typical to start with learning rate of $\alpha = 1$ and slowly shrinking it towards 0.

Temporal Difference Learning Example

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



$$V^\pi(s) = (1-\alpha)V^\pi(s) + \alpha\,(R(s,\pi(s),s') + \gamma V^\pi(s'))$$

Problems with Passive Reinforcement Learning

1. Learn the utilities value of all states under a given policy.
2. Finding an optimal policy for our agent requires knowledge of the q-utilities of states.
3. Computing q-utilities from the state utilities requires a transition function and reward function as dictated by the Bellman equation.

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

4. Passive Reinforcement Learning is used in tandem with some model-based learning to acquire estimates of T and R in order to effectively update the policy followed by the learning agent.

Active Reinforcement Learning: Q-learning

Idea of Q-learning: learning the q-utilities directly, as a result, Q-learning is entirely model-free.

Active Reinforcement Learning: Q-learning

Idea of Q-learning: learning the q-utilities directly, as a result, Q-learning is entirely model-free.

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

$$V^*(s') = \max_{a'} Q^*(s',a')$$

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

Q-learning Algorithm

Q-learning uses the following update known as q-value iteration:

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

1. Compute a sample:

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

2. Update $Q(s,a)$: $Q(s,a) = (1-\alpha)Q(s,a) + \alpha\ sample$

Q-learning

Q-learning Algorithm

1. Compute a sample:

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

2. Update $Q(s, a)$: $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha\ sample$

- As long as we spend enough time in exploration and decrease the learning rate α at an appropriate pace, Q-learning learns the optimal Q-values for every Q-state
- TD learning and direct evaluation learn the values of states under a policy by following the policy before determining policy optimality via other techniques
- Q-learning can learn the optimal policy directly by taking suboptimal or random actions.

Exploration: ϵ-greedy

1. Simplest scheme for exploration: Random actions (ϵ-greedy)
    1. Select n, a random uniform number in $[0, 1]$
    2. If $n < \epsilon$, act randomly (with probability ϵ)
    3. If $n \geq \epsilon$, act on current policy (with probability $1 - \epsilon$)

Exploration: ϵ-greedy

1. Simplest scheme for exploration: Random actions (ϵ-greedy)
   1. Select n, a random uniform number in $[0, 1]$
   2. If $n < \epsilon$, act randomly (with probability ϵ)
   3. If $n \geq \epsilon$, act on current policy (with probability $1 - \epsilon$)
2. Problems with random actions?
   1. For a large ϵ: even after learning the optimal policy, the agent will still behave mostly randomly
   2. For a small ϵ: the agent will explore infrequently, leading Q-learning to learn the optimal policy very slowly.
   3. One solution: lower ϵ over time
   4. Another solution: exploration functions

Exploration: Exploration functions

- Manually tuning ϵ is avoided by exploration functions
- Use a modified Q-value iteration update to give some preference to visiting less-visited states.

$$\textit{Regular } Q\textit{-Update}: \; Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right]$$

$$\textit{Modified } Q\textit{-Update}: \; Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} f(s',a')\right]$$

f is an exploration function, with a common choice of f is

$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$$

with k predetermined value and $N(s,a)$ is the number of times Q-state (s,a) has been visited

Exploration: Exploration functions

- A state s select the action that has the highest $f(s, a)$ from each state
- Agents never make a probabilistic decision between exploration and exploitation
- Exploration is encoded by the exploration function, since $\frac{k}{N(s,a)}$ give a "bonus" to some infrequently taken action
- This bonus decreases when As time goes states are visited more frequently state and $f(s, a)$ regresses towards $Q(s, a)$, making exploitation more exclusive.

Approximate Q-learning

1. Q-learning just stores all q-values for states in tabular form.
2. Not particularly efficient for applications of reinforcement learning with several thousands or even millions of states.

Approximate Q-learning

1. Q-learning just stores all q-values for states in tabular form.
2. Not particularly efficient for applications of reinforcement learning with several thousands or even millions of states.

Idea of approximate Q-learning:

- Represents each state as feature vector $(f_1(s,a), f_2(s,a), \ldots)$
  For example a feature vector for Pacman may encode: the distance to the closest ghost, the distance to the closest food pellet, the number of ghosts.
- Store a single weight vector to compute approximate Q-values. For example for linear Q-functions

$$Q(s,a) = w_1 \times f_1(s,a) + w_2 \times f_2(s,a) + \ldots + w_n \times f_n(s,a)$$

- Compute Q-values on-demand as needed.

Approximate Q-learning for linear Q-functions

1. Difference:

$$difference = [R(s, a, s') + \gamma \max_{a'} Q(s', a')] - Q(s, a)$$

2. Approximate Q-learning works almost identically to Q-learning, using the following update rule:

$$Approximate\ Q(s, a) : w_i = w_i + \alpha \times difference \times f_i(s, a)$$

   - Rather than storing Q-values for each state, approximate Q-learning only need to store a single weight vector and can compute Q-values on-demand as needed.
   - Significantly more memory efficient

3.

$$Exact\ Q(s, a) : Q(s, a) = Q(s, a) + \alpha \times difference$$