Markov Decision Process

1. Introduction to MDP

1.1 MDP problem

A MDP is a tuple $< S, A, P, R, \gamma >$, where:

- S: a finite set of **states**
- A: a finite set of **actions**
- P: a state **transition probability matrix**, $P[s'|s, a]$, which can be represented as a 3D matrix
- R: a **reward function**, R(s)
- γ: a **discount factor**, $\gamma \in [0, 1]$

Markov assumption: state transition probability only depends on the current state s, not the history of earlier states.

1.2 Solution to MDP problem

- The solution to a MDP problem is a **policy,** $\pi(s)$
- $\pi(s)$ is a function from state to action. It outputs an appropriate action for each state the agent is in
- **Optimal policy** π^*: a policy that generates highest expected utility
- π^* varies within the same problem with different rewards and risks

1.3 Evaluation of policy

- **Expected** utility of executing π starting from s: $U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)]$
- Utility of a state $U^{\pi^*}(s)$ is the expected sum of discounted reward of executing an **optimal policy** from s. Often called **value function**.
- Given the value function, we can compute an optimal policy as $\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s, a)U(s')$

2. Value iteration

2.1 Bellman equation

Bellman equation: the utility of a state is its immediate reward plus expected utility of mext states, given optimal action.
$U(s) = R(s) + \arg\max_a P(s'|s,a)U(s')$

2.2 Value iteration

2.2.1 Algorithm

Algorithm 1: Value Iteration

function VALUE-ITERATION(mdp, ϵ) returns a utility function ;
Input : mdp, an MDP with states S, actions A(s), transition model $P(s'|s,a)$, rewards R(s), discount γ; ϵ, the maximum error allowed in the utility of any state in an iteration
Output: U
Persistent: U, U', vectors of utilities for states in S, initially zero ; δ, the maximum change in the utility of any state in an iteration ;
repeat
  U $\leftarrow$ U'; $\delta \leftarrow 0$;
  foreach *state s in S* **do**
    $U'(s) \leftarrow R(s) + \gamma \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_t(s')$
    if $||U'(s) - U(s)|| > \delta$ **then**
      $\delta \leftarrow ||U'(s) - U(s)||$
    end
  end
until $\delta < \epsilon(1-\gamma)/\gamma$;
return U

- The **value iteration** algorithm repeatedly does **Bellman update**: $U_{t+1}(s) \leftarrow R(s) + \gamma \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_t(s')$
- Value iteration converges to the unique value function for discounted problems with $\gamma < 1$.

2.2.2 Contraction

- Bellman update $U_{t+1} \leftarrow BU_t$, where B is the Bellman update operator, is a **contraction**: $|BU - BU'| \leq \gamma |U - U'|$
  - This is because $|\arg\max_a f(a) - \arg\max_a g(a)| \leq \arg\max_a |f(a) - g(a)|$
- Repeated application of a contraction reaches a unique fixed point U, where $BU = U$ (equilibrium). For any initial state U_0:

$$\begin{aligned}|BU_t - BU| &= |BU_t - U| \\ &\leq \gamma |U_t - U| \\ &= \gamma |BU_{t-1} - U| \\ &\leq ... \\ &= ... \\ &\leq \gamma^t |U_0 - U|\end{aligned} \tag{1}$$

  - Bellman update converges exponentially
  - $|U_0 - U| \leq R_{max}/(1-\gamma)$, if U_0 is initialized to 0

$$\begin{aligned}U_t &= R_0 + \gamma R_1 + \gamma^2 R_2 + ...\gamma^t R_t \\ &\leq R_{max} + \gamma R_{max} + \gamma^2 R_{max} + ...\gamma^t R_{max} \\ &= \frac{R_{max}}{1-\gamma}\end{aligned} \tag{2}$$

  All states are bounded by $\pm \times \frac{R_{max}}{1-\gamma}$
  - If we run N iterations to get error at most ϵ, we have: $\gamma^N R_{max}/(1-\gamma) \leq \epsilon \implies N = \lceil log(R_{max}/\epsilon(1-\gamma))/log(1/\gamma) \rceil$
  - Terminal condition: $|U_{t+1} - U_t| \leq \epsilon(1-\gamma)/\gamma \implies |U_{t+1} - U| \leq \epsilon$

3. Policy iteration

- **Policy iteration** takes the advantage that utility function does not need to be highly accurate to give correct policy, e.g. if one action is clearly better than others.

- For policy iteration, begin with some initial policy π_0, alternate the following two steps:
  - **Policy evaluation**: given a policy π_i, and calculate $U_i = U^{\pi_i}$
  - **Policy improvement**: calculate a new policy π_{i+1} using one step look-ahead based on U_i
- Policy iteration terminates when there is no change in the policy. The number of policies are finite ($|A|^{|S|}$), hence it must terminates

---

Algorithm 2: Policy iteration

---

function POLICY-ITERATION(mdp) returns a policy ;
Input : mdp, an MDP with states S, actions A(s), transition model $P(s'|s,a)$
Output: π
Persistent: U, a vector of utilities for states in S, initially zero ; π, a policy vector indexed by state, initially random;
repeat
    U ← POLICY-EVALUATION(π, U, mdp);
    unchanged? ← true ;
    foreach *state s in S* **do**
        if $\arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') > \sum_{s'} P(s'|s,\pi(s))U(s')$ **then**
            $\pi(s) \leftarrow \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$;
            unchanged ← false
        end
    end
until *unchanged?*;
return π

- Policy evaluation equation is similar to Bellman update without a max operator: $U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi_i(s))U_i(s')$
  - For n states, it can be solved in $O(n^3)$ time
- For large state spaces, often do k iterations instead of converge: **modified policy iteration**

- To speed up, only pick a subset of states to do either policy improvement for updating in policy evaluation: **asynchronous policy iteration**

4. Online search

- State spaces grows exponentially with number of variables of a state.
- Value/policy iteration iterates through all states, thus runtime grows exponentially with number of variables
- To handle large state spaces:
  - **Function approximation** for the value function, e.g. linear function of features, deep neural networks, etc
  - **Online search with sampling**

4.1 Monte Carlo Tree search

- Algorithm:
  - **Selection**: The selection function is applied recursively until a leaf node is reached
  - **Expansion**: One or more nodes are created (depends on the number of next states)
  - **Simulation**: One simulated game is played
  - **Backpropagation**: The result of the game is backpropapgated in the tree
- MCTS repeatedly run trials from the current state (the root for its subtree in online search), where a trial:
  - Repeatedly select node to go to at next level util
    - target depth reached
    - selected node has not been discovered: create a new node, run a simulation using a **default policy** till a required depth
  - Back up the outcomes all the way to the root

- This is an **anytime policy**: when time is up, use the action that looks the best at the root at that time.
- A node n' at the next level is selected by applying an action a to s, then sampling the next state s' according to $P(s'|s,a)$
- The action is selected by balancing exploration with exploitation
- The estimated value $\overline{V}(n)$ at a node n is the **average return** of all the **trials** at n
  - The returned $r_t(n)$ of trial t starting from n with state s and next node n' is $R(s) + \gamma r_t(n')$
  - $\overline{V}(s) = \arg\max_a Q(s,a)$
- The estimated Q-function (action-value funciton) at n, $\overline{Q}(n,a)$ is the **average return** of all trials at n that starts with action a
  - $\overline{Q}(r,a)$ at the root r is used to select the action to take at the root.
- All values are updated in the back up operation to the root

4.2 Upper Confidence Tree

- UCT function to select action at node n:

  $\pi_{UCT}(n) = \arg\max_a \overline{Q}(n,a) + c\sqrt{\frac{ln(N(n))}{N(n,a)}}$, where $N(n)$ is the number of times the node has been visited, $N(n,a)$ is the number of trials through n with action a, and c is a constant.

- UCT will eventually converge to the optimal policy with enough trials

5. POMDP

5.1 Define POMDP

- The states are **partially observable**, and we receive some sensor information, **observation**, that can be used for state estimation. The observation/sensor mode is defined by $P(e|s)$, the probability of perceiving evidence e in state s.

- We do not know the actual state the agent is in, but we can track the probability distribution over the possible states. This is **belief state**, or belief for short.

- **Filtering**: tracking the probability distribution

> Belief state update: $b'(s') = \alpha P(e|s') \sum_s P(s'|s,a)b(s)$, α: normalizing constant. We write as $b' = FORWARD(b, a, e)$.

- The belief contains all the information necessary for the agent to act optimally: **the optimal action depends only on the agent's current belief**.

- Optimal policy can be described as a mapping $\pi^*(b)$ from belief to aciton

- A POMDP agent acts as follows:

  - Given the current belief b, execute the action $a = \pi^*(b)$
  - Receive the observation e
  - Set belief to FORWARD(b, a, e) and repeat

- POMDP can be viewed as a MDP in a belief space:

  - **Reward function** in the belief space can be defined as: $\rho(b) = \sum_s b(s)R(s)$
  - $P(b'|b, a)$ can be derived from the underlying POMDP
  - Together $P(b'|b, a)$ and $\rho(b)$ defines an **observable** MDP in belief space
  - The optimal policy for this MDP is also the optimal policy for the POMDP

5.2 Value iteration for POMDP

- A policy at a belief b_0 is a **conditional plan**. Multiple conditional plans are possible

- Consider a fixed conditional plan p:

  - Executing p from a state s will have utility $\alpha_p(s)$. Hence, executing it from a belief b will have expected utility $\sum_s b(s)\alpha_p(s)$ or $b \cdot \alpha_p$

  - For a fixed conditional plan p, value function $U_p(b) = b \cdot \alpha_p$ is a linear function of b
  - The optimal policy is to choose p with highest utility: $U(b) = \arg\max_p b \cdot \alpha_p$
    * $U(b) = \arg\max_p b \cdot \alpha_p$ is a hyperplane. The continuous belief space is divided into regions, each corresponding to a conditional plan optimal for that region. $U(b)$ is piecewise linear and convex
  - Let ρ be a depth d conditional plan with initial action a followed by depth $d-1$ subplans $p.e$ for observation e: $\alpha_p(s) = R(s) + \sum_{s'} P(s'|s,a) \sum_e P(e|s')\alpha_{p.e}(s'))$. This gives rise to the value iteration algorithm.

Algorithm 3: POMDP value iteration

function POMDP-VALUE-ITERATION(pomdp, e) returns a utility function

Input : pomdp, an POMDP with states S, actions A(s), transition model $P(s'|s,a)$, sensor model $P(e|s)$, rewards $R(s)$, discount γ

Output: U

Persistent: U, U', sets of plans p with associated utility vector α_p ;

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

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 vector computed according to the equation above ;

  $U' \leftarrow$ REMOVE-DOMINATED-PLANS(U')

until *MAX-DIFFERENCE*(U, U') $< \epsilon(1-\gamma)/\gamma$;

return U

6. *Dynamic decision network (DDN)*

- The execution of a POMDP over time can be represented as a **dynamic decision network**
  - Transition and sensor models represented by a **dynamic Bayesian network** (DBN)

- Add decision and utility modes to get DDN
- In DBN, state S_t becomes set of variables X_t and evidence/observation variables are E_t
- Action at time t is A_t, transition is $P(X_{t+1}|X_t, A_t)$, and sensor model is $P(E_t|X_t)$

- POMDP solvers need to solve two problems:
  - Belief tracking or filtering problem: given the history observed so far, what is the current belief
  - Planning problem: given the current belief, what is the optimal action to take