Markov Decision Processes

- A **Markov Decision Process (MDP)** is similar to a state transition system. It has states, actions, a transition function $T(s, a, s')$ specifying the probability an agent ends up in state s' when he takes action a from state s, a distribution over start states, and possibly a set of terminal states. It also has a **reward function** $R(s, a, s')$, which represents the reward that an agent receives for performing action a in state s and ending up in state s'. Sometimes the reward depends only on the state s and is abbreviated $R(s)$.

- A **q-state** is a (state, action) pair. From a state s, the agent chooses an action a, and then from that q-state (s, a), the (possibly nondeterministic) transition function chooses the resulting state s'.

- The utility of a state is not just the reward associated with that state, it also depends on what is going to happen in the future, so we need to compute utilities for sequences of rewards. We can define the utility of a sequence of rewards $[r_0, r_1, r_2, ...]$ as $U([r_0, r_1, r_2, ...]) = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + ...$ for some $0 \leq \gamma \leq 1$. If $\gamma \neq 1$, the sum will converge, so the utilities will be finite; this is called **temporal discounting**.

- The **value** of a state s is the total expected future utility given that the agent is currently in state s. Similarly, the **q-value** of a q-state (s, a) is the total expected future utility given that the agent is currently in state s and has just taken action a. Under an optimal policy, the value and q-value are denoted $V^*(s)$ and $Q^*(s, a)$, respectively.

- The **Bellman equations** specify the relationship between $V^*(s)$ and $Q^*(s, a)$:

$$V^*(s) = \max_a Q^*(s, a)$$

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

- If we plug the second Bellman equation into the first one, we get a recursive definition of V^*. We can approximate V^* by V_k^*, the optimal value considering only the next k timesteps: as $k \to \infty$, $V_k^* \to V^*$. So, to compute V^*, we can use **value iteration**: initialize $V_0^*(s)$ to 0 for all s, and then given V_i^*, plug it into the Bellman equations to compute V_{i+1}^*, and repeat until convergence. **Policy iteration** is a similar process but updates the policy instead of the values. Often, the policy will converge before the values do.

- In **reinforcement learning (RL)**, an agent gets feedback in the form of rewards, and he wants to learn a policy to maximize his expected utility. In passive RL, the policy is given, and the agent needs to learn the states' values from observation. In active RL, the agent has to choose actions and create a policy.
- In **temporal difference (TD) learning**, the agent estimates V directly from samples, without estimating T and R. It uses an exponentially-weighted moving average: $V_{new} = \alpha V_{sampled} + (1 - \alpha)V_{old}$ for some weight α.
- For constructing a policy, it is more useful to know Q than to know V. **Q-learning** is sample-based q-value iteration: each time the agent takes an action, he updates his Q values based on the Bellman equations.
- An agent must trade off between **exploration** (to learn a better policy) and **exploitation** (to get the most benefit out of the existing policy). In the ϵ**-greedy** method, with probability ϵ the agent chooses a random action, otherwise he acts according to his current policy. A better alternative is to have an "optimistic" utility function, which adds more utility to states that the agent knows less about.
- In many contexts, there are too many states to explore all of them and store their information separately. One solution is to encode states (or q-states) as feature vectors and then estimate a state's value as a linear function of its feature vector; this enables the agent to generalize from examples to new states that it has not seen before.

Exercises

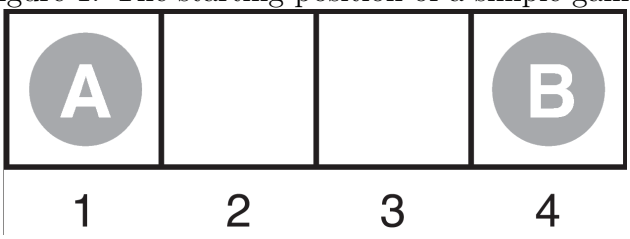
1. (AIMA 17.7) This exercise considers two-player MDPs that correspond to zero-sum, turn-taking games like those we saw last week. Let the players be A and B, and let $R(s)$ be the reward for player A in state s. (The reward for B is always equal and opposite.)

   (a) Let $U_A(s)$ be the utility of state s when it is A's turn to move in s, and let $U_B(s)$ be the utility of state s when it is B's turn to move in s. All rewards and utilities are calculated from A's point of view (just as in a minimax game tree). Write down Bellman equations defining $U_A(s)$ and $U_B(s)$.

   (b) Explain how to do two-player value iteration with these equations, and define a suitable termination criterion.

   Consider the following simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is $+1$; if player B reaches space 1 first, then the value of the game to A is -1.

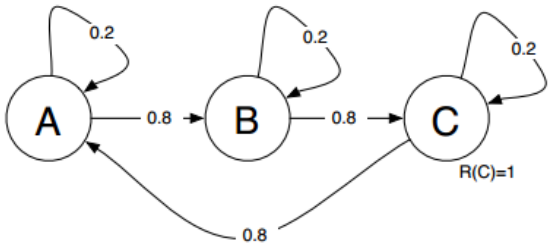
Figure 1: The starting position of a simple game.



   (c) Consider the game described in Figure 1. Draw the state space (rather than the game tree), showing the moves by A as solid lines and moves by B as dashed lines. Mark each state with $R(s)$. You will find it helpful to arrange the states (s_A, s_B) on a two-dimensional grid, using s_A and s_B as "coordinates."

   (d) Now apply two-player value iteration to solve this game, and derive the optimal policy.

2. Consider an MDP with three states, called A, B and C, arranged in a loop.



There are two actions available in each state:

- `Move`(s): with probability 0.8 moves to the next state in the loop, and with probability 0.2 stays in the same state.
- `Stay`(s): with probability 1.0 stays in the same state.

  There is a reward of 1 in state C and zero reward elsewhere. The agent starts in state A. Assume that the discount factor γ is 0.9.

  (a) Show the values of $Q(a, s)$ for 3 iterations of the TD Q-learning algorithm, given by the following equation:

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

  Let $\alpha = 1$, note the simplification that follows from this. Assume we always pick the `Move` action and end up moving to the adjacent state. That is, we see a state-action sequence: A, `Move`, B, `Move`, C, `Move`, A. The Q values start out as 0.

  (b) Characterize the weakness of Q-learning demonstrated by this example. Hint: Imagine that the chain were 100 states long.

  (c) Why might a solution based on ADP (adaptive dynamic programming) be better than Q-learning?

  (d) On the other hand, what are the disadvantages of ADP-based approaches compared to Q-learning?