Markov Decision Problems

Markov Decision Processes

Overview

A *Markov Decision Processes* (MDP) is a mathematical framework for modeling decision making under uncertainty. MDPs consist of a set of *states*, a set of *actions*, a deterministic or stochastic *transition model*, and a *reward* or *cost* function, defined below. Note that MDPs do not include observations or an explicit observation model as the environment is assumed to be fully observable at all times.

The acronym MDP can also refer to *Markov Decision Problems* where the goal is to find an optimal *policy* that describes how to act in every state of a given a Markov Decision Process. A Markov Decision Problem includes a *discount factor* that can be used to calculate the present value of future rewards and an *optimization criteria*. In *finite-horizon problems*, MDPs also include a *horizon* time that specifies when the problem ends. Strategies for minimizing cost or maximizing reward vary, and can be time dependent in finite horizon systems.

Definitions

1. **State:** $x \in \mathbb{X}$ or $s \in \mathbb{S}$. In robotics, examples of state include the pose of a rover or the configuration of a robot arm. There is typically an initial state, defined x_0 and possibly a *terminal state* that ends the problem if entered.

2. **Action:** $a \in \mathbb{A}$ or $u \in \mathbb{U}$. Examples of actions include moving to a discrete neighboring state or torques applied to a joint or wheel.

3. **Transition Model:** $x' \sim \mathcal{T}(x, a)$. We consider stochastic transitions that capture how an action a, taken from state x, will lead to state x': $\mathcal{T}(x, a) \equiv p(x' \mid x, a)$. In deterministic systems, this probability is 1 and the robot always acts as desired.

4. **Reward or Cost Function:** The reward $r(x)$ or cost $c(x)$ of taking

an action and ending in state x. The cost function is equal to the negative of the equivalent reward function and the two can be used interchangeably by swapping min and max terms during optimization. In some situations, the cost or reward is also a function of the action a, $c(x,a)$ or $r(x,a)$, or a function of the next state x' after executing action a, $r(x,a,x')$ or $c(x,a,x')$, or some even more complicated combinations like $r(x,a,x',t)$ or $p(r \mid x,a,x',t)$.

5. **Policy:** $\pi \in \Pi : \pi(x) = a$. A function that maps states into actions. This specifies how to act in any state.

6. **Horizon:** T. The number of steps used to calculate the policy. See Objective Function below.

7. **Discount Factor:** $0 \leq \gamma \leq 1$. Determines the current value of future costs or rewards. The intuition is that rewards are more valuable if they happen soon, so if a reward is received n steps in the future, it's only worth γ^n as much right now.

8. **Value Function**: $V(x,t)$. A function used to measure the expected discounted sum of rewards from following a specific policy π from state x.

9. **Objective Function:** An optimization criteria for a Markov Decision problem. Expected cumulative reward is a common objective function in reinforcement learning:

$$\mathbb{E}\left[\sum_{t=1}^{T} \gamma^t r(x_t)\right]$$

Other examples include expected infinite discounted reward:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t)\right]$$

and immediate reward:

$$\mathbb{E}\left[r(x_0)\right]$$

To disambiguate some of the notation, from this point on states will be referred to as x, transition models as $\mathcal{T}$, and horizon as T. Because we are pessimistic graduate students, we will deal in costs c, not rewards.

The goal is to choose a policy that will minimize (if we're using cost functions) or maximize (if we're using reward functions) our objective function. Remember that the policy function just describes the actions we take at each timestep, so we're effectively finding the best (on average) sequence of actions to complete our task.

Example

A successful Tetris playing algorithm, can be described as an MDP.

- **States:** Board configuration (filled/not filled), current piece. For our implementation there are approximately $2^k \times 7$ states. Note: not all configurations are valid, for example, there cannot be a piece floating in the air. This resulting in a slightly smaller number of total valid states.
- **Actions:** A policy can select any of the columns and from up to 4 possible orientations for a total of about 40 actions (some orientation and column combinations are not valid for every piece).
- **Transition Matrix:** A deterministic update of the board plus the selection of a random piece for the next time-step.
- **Cost Function:** There are several options to choose from, including: reward = +1 for each line removed, 0 otherwise; # of free rows at the top; +1 for not losing that round; etc.
- **Discount Factor:** 1
- **Time Horizon:** infinite (realistically any very large number will work, e.g. 1 billion)

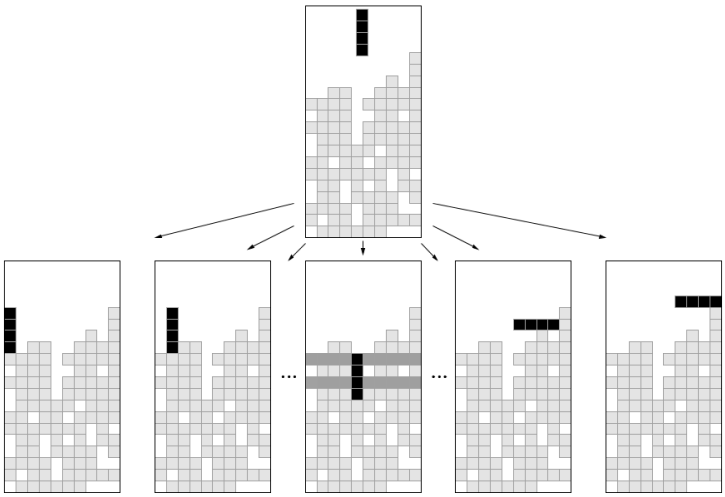

Figure 1: Example states and transitions for a Tetris scenario [].

Deterministic and Non-Deterministic MDP Algorithms

There are many approaches to solving deterministic MDPs, some better than others. Here are three possibilities:

1. The Greedy Approach: choose the action at the current time that maximizes immediate reward.

2. Exhaustive Search: explore every possible action for every possible state and choose the series of actions that maximizes the total reward.

3. Pruning: Search possible actions, but remember only the cheapest series of actions, ignoring the previously found paths with lower reward.

An exhaustive search is often the worst approach you could take. Its complexity is $O(exp(T))$.

Non-deterministic problems, where the next system state is not known with certainty, requires considering the expectation of future rewards for any given action. One strategy, called *Value Iteration*, discussed in these notes, calculates the expected sum of discounted rewards for each state (the *value* of that state). An optimal policy can then just act, by greedily selecting the action with the highest value. Some alternatives will be covered later in the course, such as *Policy Iteration* and *Q-Learning*. Policy iteration evaluates a given policy then improves upon the policy and repeats the process. Q-learning does not require a transition model, and uses samples of state-action pairs to compute the optimal action from any state.

Solving MDPs

Scenario

Let's consider the case where a robot is traversing a maze-like environment from a start location to a goal location. The environment is discretized into a 2D grid. Actions are movements in the cardinal directions. The cost is 1 for being in every state except for the goal state where the cost is 0. The goal is an absorbing state, so once we are in the goal state, we cannot leave. Our task is to choose a sequence of actions that take the robot from the start state to the goal state while minimizing the expected total cost. In other words, we want to minimize

$$\mathbb{E}\left[\sum_{t=1}^{T} c(x_t)\right]$$

We'll first look at a deterministic problem where the transition model is simply $\mathcal{T}(x, a) = 1$ if allowed, 0 otherwise. There may be obstacles or walls in the grid, in which case the robot is unable to transition into those states. In this simple deterministic problem, with the cost for each state except for the goal being 1, the value at each state is

simply the minimum number of states traversed to get from that state to the goal. The policy returned at each cell is then the direction the robot should travel to minimize its distance to the goal.

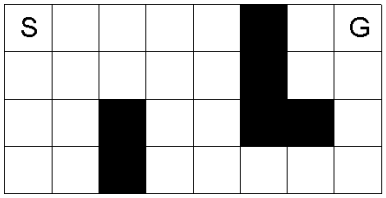

Figure 2: Discrete World, Start (S), Goal (G). Obstacles are denoted by the black squares

Recursive Formulation for Solving Deterministic MDPs

Time T − 1:

In the recursive formulation of this problem, it's easiest to start at the next-to-last timestep, $t = T - 1$. Here, the optimal policy is just choosing the action with the minimal cost and the value function at each state is the minimum cost of all actions from a given location. In the case where the cost function is independent of a ($c = c(x)$ instead of $c = c(x, a)$), the action itself has no cost and therefore any action is acceptable:

$$\pi(x, T-1) = \underset{a}{\operatorname{argmin}}\, c(x)$$

$$V(x, T-1) = \min_{a} c(x)$$

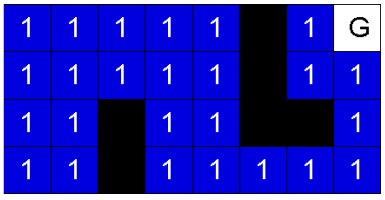

Figure 3: Value Function for each state at time T-1

Time T − 2:

Now the costs are the same everywhere except at the goal. Next, we take the action that has the minimum cost for the next state, plus the cost of being in our current state. The value of each state is the minimum of the cost of the current state plus the previous value from the next state.

$$\pi(x, T-2) = \underset{a}{\operatorname{argmin}}\ \left[c(x) + c(\mathcal{T}(x, a))\right]$$

$$V(x, T-2) = \min_{a}\ \left[c(x) + V(x', T-1)\right]$$

Time T − 3 and below

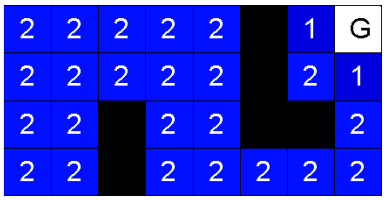

Figure 4: Value Function for each state at time T-2

We can define a general recursion to calculate the value and policy functions. For any given time t, we have:

$$\pi(x,t) = \underset{a}{\operatorname{argmin}} \ [c(x) + V(\mathcal{T}(x,a), t+1)]$$

$$V(x,t) = \min_{a} \ [c(x) + V(\mathcal{T}(x,a), t+1)]$$

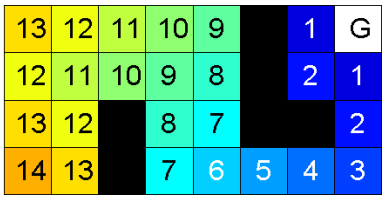

Figure 5: Final value function after T steps of Value Iteration

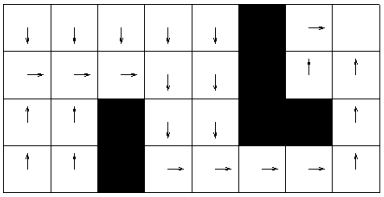

Figure 6: Action at each location using the final policy.

Following the equations above, we can write recursive algorithms that produce the best value and the best policy for any state, at any time t, considering a T-length time horizon. Algorithm 1 below describes the recursive method that computes the best value function (cost-to-go) for a given state x starting at time t and stopping at time $T-1$.

Algorithm `OptimalValue(`x,t,T`)`
  if $t = T-1$ **then**
    return $c(x)$
  end
  else
    return $\min_a c(x)$ + `OptimalValue(`$\mathcal{T}(x,a), t+1, T$`)`
  end

Algorithm 1: Recursive algorithm for computing the optimal value function

How do we compute the best policy? Let's first define an auxiliary algorithm that returns the value function with time horizon T for a

given policy π, starting at state x. This is called *policy evaluation* and is described in Algorithm 7.

Algorithm `Value`(x,π,t,T)
  if $t = T-1$ **then**
    return $c(x)$
  end
  else
    return $c(x) +$ `Value`$(\mathcal{T}(x,\pi(x)),\pi,t+1,T)$
  end

Algorithm 2: Policy evaluation: a recursive algorithm that computes the value function for a given policy

Next, we define the algorithm that uses Algorithm 7 for computing the best policy $\pi^*(x,t)$ for all states and time steps:

Algorithm `OptimalPolicy`(x,π,t,T)
  for t = *T-1*, ..., *0* **do**
    for $x \in \mathbb{X}$ **do**
      if *t=T-1* **then**
        $\pi^*(x,t) = \underset{a}{\operatorname{argmin}}\ c(x)$
      end
      else
        $\pi^*(x,t) =$
        $\underset{a}{\operatorname{argmin}}\ c(x)\ +$ `Value`$(\mathcal{T}(x,a),\pi^*,t+1,T)$
      end
    end
  end

Algorithm 3: Algorithm for computing the optimal policy

Note that the complexity of computing the value function recursively is $O(|\mathbb{X}||\mathbb{A}|T^2)$. However, because we are repeatedly calculating many of these function calls, you can *memoize* previously computed value functions (i.e. from future time steps, since the algorithm moves from future to past) resulting in an algorithm with complexity $O(|\mathbb{X}||\mathbb{A}|T)$.

One important concept we can observe from the algorithms above is that if we use the value function we never need to explicitly compute the policy. Polciy and value are *not* the same, but if the value function is given, the policy can be easily discovered, as shown below:

$$\pi(x,t) = \underset{a}{\operatorname{argmin}}\ [c(x) + V(\mathcal{T}(x,a),t+1)]$$

with

$$V(x,t) = \min_{a}\ [c(x) + V(\mathcal{T}(x,a),t+1)]$$

Note about finite horizon problems: When we have a finite horizon, the value function is also a function of time. This scenario is similar to a hockey game, in which a team's actions may vary widely depending on the time remaining. If a team is losing and there are seconds left, they may choose to pull their goalie off the ice and have an extra player. If they are at the start of the game and losing, pulling the goalie is a very poor decision.

Recursive Formulation for Solving Non-deterministic MDPs

Consider non-deterministic MDPs with uncertainty in the transition model. Here we will take the expectation over the value function:

$$\begin{aligned}
\pi(x,t) &= \underset{a}{\operatorname{argmin}} \left[c(x) + \mathbb{E}\left[V(x',t+1)\right]\right] \\
&= \underset{a}{\operatorname{argmin}} \left[c(x) + \sum_{x'} p(x'|a,x)V(x',t+1)\right] \\
V_t(x,t) &= \min_a \left[c(x) + \mathbb{E}\left[V(x',t+1)\right]\right] \\
&= \min_a \left[c(x) + \sum_{x'} p(x'|a,x)V(x',t+1)\right]
\end{aligned}$$

This approach now has complexity $O(|\mathbb{X}|^2|\mathbb{A}|T)$. However, since we don't have to sum over all $x \in \mathbb{X}$ in most cases, this typically reduces to $O(k|\mathbb{X}||\mathbb{A}|T)$, where k is the average number of neighbouring states. If our environment is continuous, the sums above become integrals as we are integrating over the state space.

Infinite Horizon Problems

As T approaches infinity, the value function for every state will often converge to a fixed value (or, equivalently, for every state, the policy will converge to a given action). In some cases this is not assured. Typically, failure of convergence for the infinite horizon problem is caused by divergence (for example, when the goal is unreachable), but oscillation of the value function can also prevent the value function from converging. A simple example of the oscillation problem is shown below:

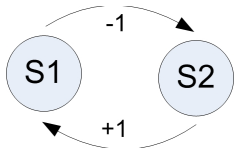

Figure 7: Value Function Oscillation

If the value function does converge, we are assured a policy that is optimal.

Rewards and Discount Factors

Thus far, we have only talked about cost functions in our examples. Now imagine that we are instead using a reward function, where the robot gets zero points for each move, unless it moves into the goal, in which case it gets 100 points. You can see that there is very little urgency for the robot to move towards the goal, as it can spend as many steps as it wants wandering the state space before reaching the goal.

In order to avoid situations like this, we can apply the discount factor mentioned above. Since discount factors value obtaining rewards sooner, rather than later, they would require the robot to move to the goal as quickly as possible to be optimal. Discount factors can alternatively be thought of as a way of contending with the possibility of death. Under this interpretation, at each time step, the robot lives with probability γ, and dies with probability $(1-\gamma)$ (goes to an absorbing state that has 0 reward or value). The value function then becomes:

$$\begin{aligned} V(x,t) &= \min_a \left[c(x) + \sum_{x'} \left[\gamma \left[p(x'|a,x) V(x', t+1) \right] + (1-\gamma) * 0 \right] \right] \\ &= \min_a \left[c(x) + \gamma \sum_{x'} p(x'|a,x) V(x', t+1) \right] \end{aligned}$$

The fixed point version of the above equation is called the **Bellman equation**.

$$V(x) = \min_a \left[c(x) + \gamma \sum_{x'} p(x'|a,x) V(x') \right]$$

We will explore this equation in more detail below.

Convergence and Optimal Solutions

If $\gamma < 1$ we can guarantee with probability 1 that the value function will converge. For some special cases, the function can converge for $\gamma = 1$, but this is not generally true.

It is important to keep in mind that once the value converges, it becomes invariant with relation to the time.

$$V(x,t) \xrightarrow{t \to \infty} V(x) = \min_a \left[c(x) + \gamma \sum_{x'} p(x'|x,a) V(x') \right]$$

And the same happens for the optimal policy:

$$\pi(x,t) \xrightarrow{t \to \infty} \pi(x) = \arg\min_a \left[c(x) + \gamma \sum_{x'} p(x'|x,a) V(x') \right]$$

There are two iterative approaches for finding this convergence value.

Approach 1 In this approach, we define small threshold ε (this could be interpreted as a as a confidence level) and we will run the algorithm for a time horizon that is sufficiently large so that the oscillation of that value will be of magnitude $O(\varepsilon)$. Thus, we will choose T such that $\gamma^T = O(\varepsilon)$, i.e. $T = O(log(\varepsilon))$.

Data: convergence precision: ε
Pick T s.t. $T = \mathcal{O}(\log(\varepsilon))$
Algorithm `Value(`x,t,T`)`
  if $t = T - 1$ **then**
    return $c(x)$
  end
  else
    return $\min_a c(x) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,a)$ `Value(`$x', t+1, T$`)`
  end

Algorithm 4: Recursive algorithm with convergence threshold

This algorithm provides the value function for a given state x. To obtain the value function for *all* states, we execute:

Data: convergence precision: ε
for $x \in \mathbb{X}$ **do**
  $V(x) =$`Value(`x,o,T`)`
end
return $V(x), \forall x$

Approach 2 The second approach uses an iterative method, based on the Bellman equation, where the result obtained in one step is plugged back into the equation until it converges.

for $x \in \mathbb{X}$ **do**
  $V(x) = \min_a c(x)$
end
while *does not converge* **do**
  for $x \in \mathbb{X}$ **do**
    $V^{new}(x) = \min_a c(x) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,a) V^{old}(x')$
  end
  $V^{old}(x) \leftarrow V^{new}(x), \forall x$
end
return $V^{new}(x), \forall x$

Algorithm 5: Iterative approximation algorithm

Both algorithms will return the optimal value function for all states. As mentioned earlier, if the value function is known, it is possible to obtain the policy. Thus, these algorithms also allow us to

obtain the optimal policy for every state.

Approach 1 will theoretically give better results, since it is actually the optimal solution for the finite horizon problem. Approach 2 is not the solution for *any* specific problem (it is an approximate iterative method), at least not until the moment it converges (in this case, it is the solution for the Bellman equation). Nevertheless, Approach 1 can be costly: it requires a considerable amount of extra memory, since it keeps track of *all* future values on any given time step. Approach 2 initializes the value function V and iteratively finds better approximations of that value by plugging its current value into the solution equation. Compared with the first approach, this approach gives results that are slightly inferior, but requires a much smaller amount of memory.

Related Reading

- Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. Probabilistic Robotics. Cambridge, MA: MIT, 2005. Ch 14, pp 499-502 for most relevant material.
- Andrew Moore's slides: http://www.autonlab.org/tutorials/mdp.html
- Boumaza, A. How to design good Tetris players, Tech Report, University of Lorraine, LORIA, 2014.