1
Markov Decision Problems

1.1 *Markov Decision Processes*

Overview

We require a formal model of decision making to be able to synthesize and analyze algorithms. In general, making an "optimal" decision requires reasoning about the entire history previous observations, even with perfect knowledge of how an environment works.

A powerful notion that comes to us from the physical sciences is the idea of *state* — a sufficient statistic to predict the future that renders it independent of the past. In classical mechanics, the phase space of positions and momenta forms that state: together with the knowledge of an isolated rigid body (it's inertia) and any torques applied, we can predict the future pose of the object without knowledge of the past.

A *Markov Decision Process* (MDP) is a mathematical framework for modeling decision making under uncertainty that attempts to generalize this notion of a *state* that is sufficient to insulate the entire future from the past. 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: in other words, an agent can observe the state of the world.

The acronym MDP can also refer to a *Markov Decision Problem* 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 criterion*. 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 should be time-dependent in finite horizon systems.

The key property – indeed the eponymous property – of an MDP is that it is *Markov*. That is, the probability of observing future states given the past depends (and given a fixed sequence of actions) only the most recent state and is conditionally independent of the full history. We make that more precise after we introduce notion below.

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. State is meant to invoke the notion of a full description (like position and velocity in classical mechanics) of the system under consideration that makes the previous trajectory irrelevant to the prediction of the future.

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:** For stochastic systems, we represent the transition model as the probability of an action a, taken from state x, leading to state x', denoted $x' \sim \mathcal{T}(x, a)$. Here $\mathcal{T}$ can be a probability mass function in case of systems with discrete set of states or a probability density function if the system has a continuous set of states. In deterministic systems, we often explicitly denote the transition model as a deterministic function, i.e., $x' = \mathcal{T}(x, a)$. Note, however, that it is also possible to realize deterministic systems with a stochastic model with the Dirac delta distribution. In an MDP this distribution is well defined, and independent of the past: $p(x'|x, a, \textit{history of all previous x's and a's}) = \mathcal{T}(x, a)$.

4. **Reward or Cost Function:** The reward $r(x, a)$ or cost $c(x, a)$ of taking an action a at a state x. The reward and the cost function can be used interchangeably: we can get the same solution if we define the cost function as the negative of a given reward function and switch the max (as for reward) to min (as for cost) during optimization. In some situations, the cost or reward can be a function of only the state s, i.e., $r(x)$ or $c(x)$, 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 being also a function of time, i.e., $r(x, a, x', t)$, or can itself be a random variable (i.e., with distribution $p(r \mid x, a, x', t)$). The last form is the most general form that obeys the Markov property and enables efficient computation.

5. **Horizon:** $T \in \mathbb{N}$. The process is considered ended after T steps.

This often encodes the number of steps that we care/are able to execute the policy. See Objective Function below. [1]

6. **Discount Factor:** $0 \leq \gamma \leq 1$. It 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.

7. **Policy:**$\pi \in \Pi : \pi(x,t) = a$. A function that maps states (and an optional time step) into actions. This specifies how to act in any state. [2]

8. **Value Function**: $V^{\pi}(x,t)$. A function used to measure the expected discounted sum of rewards from following a specific policy π from state x. The optimal value function, denoted $V^*(x,t)$, is the value function of the optimal policy π^*, i.e. the policy that yields the highest value for each state x.

9. **Objective Function:** An optimization criteria for a Markov Decision problem.[3] Expected cumulative reward is a common objective function in reinforcement learning:

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t \, r(x_t, a_t)\right]$$

Other examples include expected infinite discounted reward:

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

and immediate reward:

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

[1] If $T = 1$, optimal control can be reduced to a greedy search, that is choosing the action with the highest reward. If $0 < T < \infty$, then one must reason T steps ahead to determine the optimal policy starting from the initial state. Often there is no discount factor and the optimal policy may vary wildly as a function of time. The case where $T = \infty$ is typically more likely to converge, as a discount factor γ is used to dampen the effects of oscillation or any time-dependent properties.

[2] In the simplest case, a policy is simply a map from the current state to an action, but policies can be much more general and include information about the transition model or information about the history of previous states ($\pi : \{x_0, ..., x_t\} \times \mathcal{T} \rightarrow \mathbb{A}$). We can show that if a decision problem is Markovian, an optimal policy need only be a function of state and time, rather than further history.

[3] Note that optimizing such an objective function does not require the Markov property – that property helps us find policies efficiently.

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 action we take at each time step, so we're effectively finding the best (on average) sequence of actions to complete our task.

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 academics, we will deal in costs c, not rewards.

Example

Consider the simplified game of *Tetris*, where randomly falling pieces must be placed on the game board. Each horizontal line completed is cleared from the board and scores points for the player. The game terminates when the board fills up. The game of Tetris can be modeled as a Markov Decision Process.

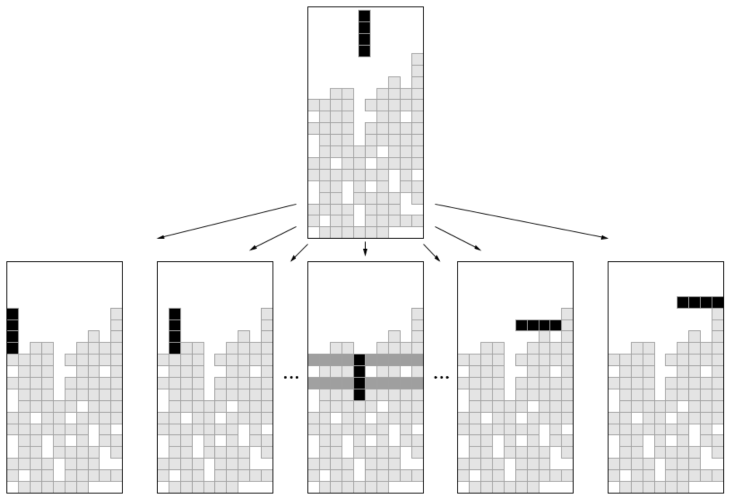


Figure 1.1.1: Example states and transitions for a Tetris scenario with figure from [3].

- **States:** Board configuration (each of k cells can be filled/not filled), current piece (there are 7 pieces total). In this implementation, there are therefore 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 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.

Deterministic and Non-Deterministic MDP Algorithms

For Deterministic MDPs the transition model is deterministic or, equivalently, we know with certainty what the next state x' will be given the current state x and the action a. Solving deterministic MDPs is often traditionally posed as a search problem. There are many approaches to solving deterministic MDPs using search, many of which are much more efficient than generic MDP approaches. Here are three flavors of approach that one might try:

1. The Greedy Approach: choose the action at the current time that minimizes the immediate cost.

2. Naive Exhaustive Search: explore every possible action for every possible state and choose the series of actions that minimizes the total cost.

3. Pruning: Search possible actions, but remember only the cheapest series of actions, ignoring the previously discovered paths with higher cost.

A naive exhaustive search is often computationally ineffective as its complexity is $O(\exp(T))$.

An exhaustive search can produce the optimal policy at the expensive of high computational (and sample complexity) cost. While the greedy approach is often cheap to compute, it may sometimes produce policies that are not remotely good. The pruning approach balances the computational cost and the quality of the resulting policies. Often it can produce a reasonably good policy in a much shorter time compared to the exhaustive search algorithms. However, if we care to find *the optimal policy*, then we need to consider all policies.

Non-deterministic problems, where the next system state is not known with certainty, naturally suggest considering the expectation of future rewards for any given action. One strategy, called *Value Iteration*,[4] discussed in the later section, calculates the expected sum of discounted rewards for each state under the optimal policy (the *value* of that state, denoted V^*, also known as the optimal value function) without explicitly computing the optimal policy. 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.

[4] The Value Iteration algorithm is also applicable to deterministic MDPs. In fact, we will see how to use Value Iteration to solve deterministic MDPs in the next section.

1.2 *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 a terminal "absorbing" state, so once we are in the goal state, we cannot leave – we have achieved nirvana and the suffering is over. 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

$$\mathbb{E}\left[\sum_{t=0}^{T-1} c(x_t, a_t)\right]$$

We'll first look at a deterministic problem where the robot will move to the adjacent cell in the direction of the action if the cell is free: 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 optimal value at each state is simply the minimum number of states traversed to get from that state to the goal. The optimal policy returned at each cell is then the direction the robot should travel to minimize the number of steps needed to reach the goal.

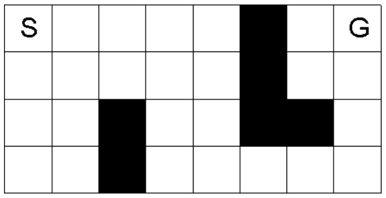


Figure 1.2.1: Discrete World, Start (S), Goal (G). Obstacles are denoted by the black squares

Recursive Formulation for Solving Deterministic MDPs

Time T − 1:

We can write this in a straightforward recursive formulation of this problem, we start at the 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.

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

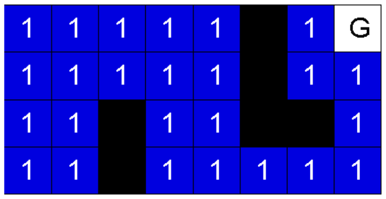


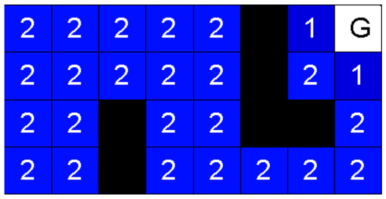
Figure 1.2.2: Optimal Value Function for each state at time $T - 1$

Time T − 2:

Now the values at the last timestep are the same everywhere except at the goal. Next, consider the next-to-last step $t = T - 2$. Suppose that we are at state x and we take action a, the total cost would be the value of the next state $x' = \mathcal{T}(x, a)$ at the last timestep $T - 1$

plus the immediate cost of taking action a in our current state x. Therefore, we should simply choose an action a that minimizes the sum these two terms. The optimal value of each state is then the minimum of the cost of the action a at current state x and plus the optimal value of the next state x' at the last timestep $T-1$.

$$\pi^*(x, T-2) = \underset{a}{\operatorname{argmin}} \left[c(x,a) + V^*(x', T-1)\right]$$
$$V^*(x, T-2) = \min_a \left[c(x,a) + V^*(x', T-1)\right]$$



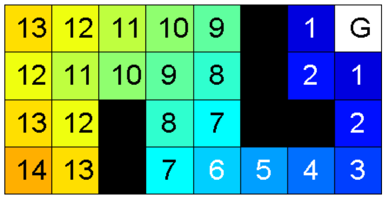
2	2	2	2	2		1	G
2	2	2	2	2		2	1
2	2		2	2			2
2	2		2	2	2	2	2

Figure 1.2.3: Optimal Value Function for each state at time $T-2$

Time T − 3 and below

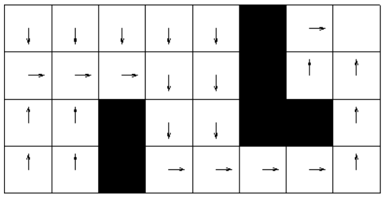
We can define a general recursion to calculate the optimal value and optimal policy functions. For any given time $t \leq T-2$, we have:

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



13	12	11	10	9		1	G
12	11	10	9	8		2	1
13	12		8	7			2
14	13		7	6	5	4	3

Figure 1.2.4: Final value function after T steps of Value Iteration



↓	↓	↓	↓	↓		→	
→	→	→	↓	↓		↑	↑
↑	↑		↓	↓			↑
↑	↑		→	→	→	→	↑

Figure 1.2.5: Action at each location using the final policy.

Following the equations above, we can write recursive algorithms that produce the optimal value and the optimal 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 $\min_a c(x,a)$
    end
    else
        return $\min_a c(x,a) + \texttt{OptimalValue}(\mathcal{T}(x,a), t+1, T)$
    end
```

Algorithm 1: Recursive algorithm for computing the optimal value function

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

$$\pi^*(x,t) = \operatorname*{argmin}_a \left[c(x,a) + V^*(\mathcal{T}(x,a), t+1)\right].$$

But what if we want to get the optimal policy *while* computing the optimal value? 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 2.

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

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

The above can, of course, be implemented as an in-place dynamic program by starting from the last time-step as in Algorithm 3.

We can also extract via a *dynamic program* (backwards induction) that proceeds from the last time step Algorithm 2 to compute the optimal policy $\pi^*(x,t)$ for all states and time steps:

```
Algorithm OptimalPolicy($x$, $t$, $T$)
    for $t = T-1, \ldots, 0$ do
        for $x \in \mathbb{X}$ do
            if $t = T-1$ then
                $\pi^*(x,t) = \operatorname{argmin}_a c(x,a)$
            end
            else
                $\pi^*(x,t) =$
                  $\operatorname{argmin}_a c(x,a) +$ 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 via dynamic programming is $O(|\mathbb{X}||\mathbb{A}|T^2)$. However, because we are repeatedly calculating many of these function calls, we can *memoize* previously computed value functions (i.e. from future time steps) resulting in an algorithm with complexity $O(|\mathbb{X}||\mathbb{A}|T)$. Below, we'll explictly use backwards induction to create *Value Iteration*, the "industry standard" efficient means to compute the optimal value function rather than rely on ad-hoc memoization.

It is worth noting that when a decision problem has 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 scoring player. At the start of the game, even if losing, pulling the goalie is generally a very unwise decision.

Backwards Induction Formulation for Solving General MDPs

Consider now non-deterministic MDPs– that is, problems with uncertainty in the transition model. Here we will consider optimizing the expectation over the optimal value function:

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

Applying backwards induction (dynamic programming) instead of a recursive formulation, we get what is known as *Value Iteration*:

```
Algorithm OptimalValue($x$, $t$, $T$)
    for $t = T-1, \dots, 0$ do
        for $x \in \mathbb{X}$ do
            if $t = T-1$ then
                $V(x,t) = \min_a c(x,a)$
            end
            else
                $V(x,t) = \min_a c(x,a) + \sum_{x' \in \mathbb{X}} p(x'|x,a) V(x,t+1)$
            end
        end
    end
```

Algorithm 4: Dynamic Programming Value Iteration for computing the optimal value function.

This approach now has complexity $O(|\mathbb{X}|^2|\mathbb{A}|T)$. However, since we often don't have to sum over all $x \in \mathbb{X}$ as the probability of transitioning to those states may be 0, this typically reduces to $O(k|\mathbb{X}||\mathbb{A}|T)$, where k is the average number of neighbouring states. In a deterministic problem, of course $k = 1$. If our environment is continuous, the sums above become integrals as we are integrating over the state space.

Infinite Horizon Problems

Recall that when we have a finite horizon, both the optimal value function and the optimal policy are functions of time. However, as T approaches infinity, we expect that the optimal value function and the optimal policy no longer have such dependence on time. Consider, for example, the maze problem above: we would expect the value function to stabilize as the horizon T gets large. Similarly, it would seem surprising to alter our policy at different time steps when there is no time limit (imagine a game that lasts forever).

Exercise: Construct examples that lead to value function divergence. Relate to the classical convergence criteria for series in sequences in college-level calculus.

In some cases, the value function (optimal, or for a given policy) will **not** converge in the infinite horizon case. 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:

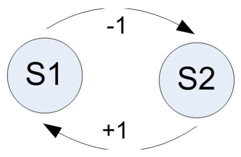


Figure 1.2.6: Value Function Oscillation

If the value function does converge, we are assured a stationary feedback policy that is optimal. [5]

[5] Exercise: Why? Make the argument.

Rewards and Discount Factors

Thus far, we have only talked about cost functions in our examples. Instead, imagine using a **reward** function, where the robot gets zero points for each move, unless it moves into the goal, in which case it receives 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 while still receiving the same 100 points.

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 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 optimal value function then becomes:

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

The fixed point version of the above equation (i.e., what we would expect to hold as the finite horizon value function to converge as $T\to\infty$) is called the **Bellman equation**.

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

We will explore this equation in more detail below.

Convergence and Optimal Solutions

If $\gamma < 1$, we can guarantee that the sum of rewards achieved by the agent is finite with probability 1 (assuming the reward is as well for each state and time) and that the optimal value function will converge. For many special cases, the value function will also converge for $\gamma = 1$, but this is not generally true for the reasons we discussed above.

It is important to bear in mind that once the value converges, it– and the optimal policy– becomes invariant with relation to the time.[6]

[6] Exercise: Convince yourself this must be true.

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

And the same happens for the optimal policy:

$$\pi^*(x,t) \xrightarrow{t\to\infty} \pi^*(x) = \operatorname*{argmin}_a \left[c(x,a) + \gamma \sum_{x'} p(x'|x,a)\, V^*(x') \right]$$

There are two iterative approaches for finding this convergence value.

Approach 1 In this approach, we define a 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 error in that value will be of magnitude $O(\varepsilon)$. Choosing T such that $\gamma^T = O(\varepsilon)$, i.e. $T = O(\log(\frac{1}{\varepsilon}))$, ensures that our error is $O(\varepsilon)$. We then simply run Algorithm 4 for T time-steps, use execute the resulting (time-varying!) policy. [7]

[7] It's unclear what to do in this approach when the policy executes T or more steps. Cycling the policy again could be a reasonable procedure but is ad-hoc. Of course, theoretically it doesn't matter because times larger than T, by construction, are exponentially damped in their significance.

```
Algorithm OptimalValue($x$, $t$, $T$)
    for $t = T-1, \ldots, 0$ do
        for $x \in \mathbb{X}$ do
            if $t = T-1$ then
                $V(x,t) = \min_a c(x,a)$
            end
            else
                $V(x,t) = \min_a c(x,a) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,a) V(x,t+1)$
            end
        end
    end
```

Algorithm 5: Dynamic Program for creating an optimal value function on the infinite horizon by finite horizon approximation

Approach 2 Alternately one can use an iterative, in-place 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
    $V(x) = \min_a c(x,a)$
end
while does not converge do
    for $x \in \mathbb{X}$ do
        $V^{new}(x) = \min_a c(x,a) + \gamma \sum_{x' \in \mathbb{X}} p(x'|x,a)\, V^{old}(x')$
    end
    $V^{old}(x) \leftarrow V^{new}(x), \forall x$
end
return $V^{new}(x), \forall x$
```

Algorithm 6: Iterative approximation algorithm

Both algorithms will return the optimal value function for all states as the number of iterations tends to infinity. As mentioned earlier, once 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 can be demonstrated to have theoretically stronger performance bounds if we execute the time-varying policy that results rather than keeping only the value and policy computed at $t = 0$, perhaps intuitively as it is actually the optimal solution for the finite horizon problem.[8] Approach 2 is not the optimal solution for *any* specific problem (it is an approximate iterative method). Nevertheless, Approach 1 can be costly: it requires a considerable amount of extra memory, since it keeps track of *all* future values for each 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 has a slower convergence rate as a function of the number of iterations in the worst case, but requires a smaller amount of memory. One can also consider simple variants (covered in [Puterman, 1994]) that maintain a single value functions and update data *in place*.[9]

[8]

[9] Similar to a *Gauss-Seidel* method.

1.3 *Related Reading*

[1] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. Probabilistic Robotics. Cambridge, MA: MIT, 2005. Ch 14, pp 499-502 for most relevant material.

[2] Andrew Moore's slides: `http://www.autonlab.org/tutorials/mdp.html`

[3] Boumaza, A. How to design good Tetris players, Tech Report, University of Lorraine, LORIA, 2014.

[4] Puterman, M. Markov Decision Processes: Discrete Stochastic Dynamic Programming, 2005.

2
Bibliography

M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.