Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Markov Decision Processes

We've said before that this book is about "sequential decisioning" under "sequential uncertainty." In Chapter **??**, we covered the "sequential uncertainty" aspect with the framework of Markov Processes, and we extended the framework to also incorporate the notion of uncertain "Reward" each time we make a state transition - we called this extended framework Markov Reward Processes. However, this framework had no notion of "sequential decisioning." In this chapter, we further extend the framework of Markov Reward Processes to incorporate the notion of "sequential decisioning," formally known as Markov Decision Processes. Before we step into the formalism of Markov Decision Processes, let us develop some intuition and motivation for the need to have such a framework - to handle sequential decisioning. Let's do this by re-visiting the simple inventory example we covered in Chapter **??**.

1.1 Simple Inventory Example: How much to Order?

When we covered the simple inventory example in Chapter **??** as a Markov Reward Process, the ordering policy was:

$$\theta = \max(C - (\alpha + \beta), 0)$$

where $\theta \in \mathbb{Z}_{\geq 0}$ is the order quantity, $C \in \mathbb{Z}_{\geq 0}$ is the space capacity (in bicycle units) at the store, α is the On-Hand Inventory and β is the On-Order Inventory ((α, β) comprising the *State*). We calculated the Value Function for the Markov Reward Process that results from following this policy. Now we ask the question: Is this Value Function good enough? More importantly, we ask the question: Can we improve this Value Function by following a different ordering policy? Perhaps by ordering less than that implied by the above formula for θ? This leads to the natural question - Can we identify the ordering policy that yields the *Optimal* Value Function (one with the highest expected returns, i.e., lowest expected accumulated costs, from each state)? Let us get an intuitive sense for this optimization problem by considering a concrete example.

Assume that instead of bicycles, we want to control the inventory of a specific type of toothpaste in the store. Assume you have space for 20 units of toothpaste on the shelf assigned to the toothpaste (assume there is no space in the backroom of the store). Asssume that customer demand follows a Poisson distribution with Poisson parameter $\lambda = 3.0$. At 6pm store-closing each evening, when you observe the *State* as (α, β), you now have a choice of ordering a quantity of toothpastes from any of the following values for the order quantity $\theta : \{0, 1, \ldots, \max(20 - (\alpha + \beta), 0)\}$. Let's say at Monday 6pm store-closing, $\alpha = 4$ and $\beta = 3$. So, you have a choice of order quantities from among the integers in the range of 0 to (20 - (4 + 3) = 13) (i.e., 14 choices). Previously, in the Markov Reward Process model, you would have ordered 13 units on Monday store-closing. This means on Wednesday morning at 6am, a truck would have arrived with 13 units of the toothpaste. If you sold say 2 units of the toothpaste on Tuesday, then on Wednesday 8am at store-opening, you'd have 4 + 3 - 2 + 13 = 18 units of toothpaste on your shelf. If you keep

following this policy, you'd typically have almost a full shelf at store-opening each day, which covers almost a week worth of expected demand for the toothpaste. This means your risk of going out-of-stock on the toothpaste is extremely low, but you'd be incurring considerable holding cost (you'd have close to a full shelf of toothpastes sitting around almost each night). So as a store manager, you'd be thinking - "I can lower my costs by ordering less than that prescribed by the formula of $20 - (\alpha + \beta)$." But how much less? If you order too little, you'd start the day with too little inventory and might risk going out-of-stock. That's a risk you are highly uncomfortable with since the stockout cost per unit of missed demand (we called it p) is typically much higher than the holding cost per unit (we called it h). So you'd rather "err" on the side of having more inventory. But how much more? We also need to factor in the fact that the 36-hour lead time means a large order incurs large holding costs *two days later*. Most importantly, to find this right balance in terms of a precise mathematical optimization of the Value Function, we'd have to factor in the uncertainty of demand (based on daily Poisson probabilities) in our calculations. Now this gives you a flavor of the problem of sequential decisioning (each day you have to decide how much to order) under sequential uncertainty.

To deal with the "decisioning" aspect, we will introduce the notion of *Action* to complement the previously introduced notions of *State* and *Reward*. In the inventory example, the order quantity is our *Action*. After observing the *State*, we choose from among a set of Actions (in this case, we choose from within the set $\{0, 1, \ldots, \max(C - (\alpha + \beta), 0)\}$). We note that the Action we take upon observing a state affects the next day's state. This is because the next day's On-Order is exactly equal to today's order quantity (i.e., today's action). This in turn might affect our next day's action since the action (order quantity) is typically a function of the state (On-Hand and On-Order inventory). Also note that the Action we take on a given day will influence the Rewards after a couple of days (i.e. after the order arrives). It may affect our holding cost adversely if we had ordered too much or it may affect our stockout cost adversely if we had ordered too little and then experienced high demand.

1.2 The Difficulty of Sequential Decisioning under Uncertainty

This simple inventory example has given us a peek into the world of Markov Decision Processes, which in general, have two distinct (and inter-dependent) high-level features:

- At each time step t, an *Action* A_t is picked (from among a specified choice of actions) upon observing the *State* S_t
- Given an observed *State* S_t and a performed *Action* A_t, the probabilities of the state and reward of the next time step (S_{t+1} and R_{t+1}) are in general a function of not just the state S_t, but also of the action A_t.

We are tasked with maximizing the *Expected Return* from each state (i.e., maximizing the Value Function). This seems like a pretty hard problem in the general case because there is a cyclic interplay between:

- actions depending on state, on one hand, and
- next state/reward probabilities depending on action (and state) on the other hand.

There is also the challenge that actions might have delayed consequences on rewards, and it's not clear how to disentangle the effects of actions from different time steps on a

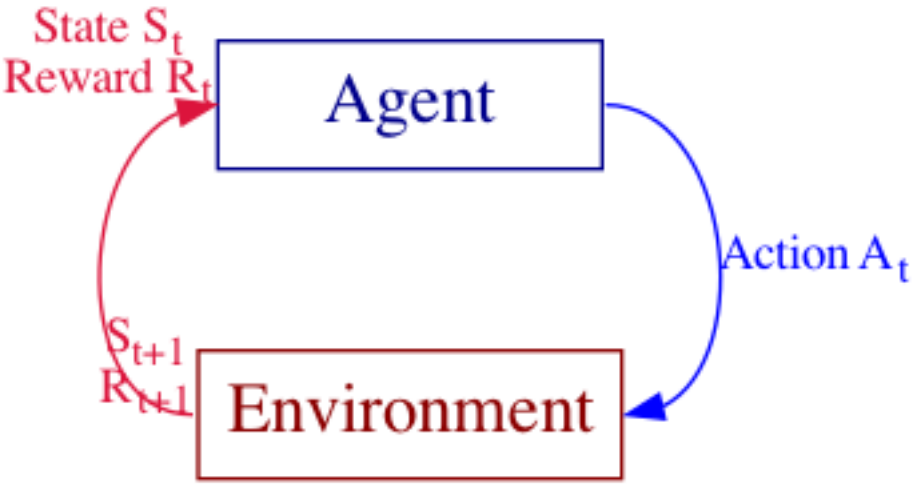

Figure 1.1: Markov Decision Process

future reward. So without direct correspondence between actions and rewards, how can we control the actions so as to maximize expected accumulated rewards? To answer this question, we will need to set up some notation and theory. Before we formally define the Markov Decision Process framework and it's associated (elegant) theory, let us set up a bit of terminology.

Using the language of AI, we say that at each time step t, the *Agent* (the algorithm we design) observes the state S_t, after which the Agent performs action A_t, after which the *Environment* (upon seeing S_t and A_t) produces a random pair: the next state state S_{t+1} and the next reward R_{t+1}, after which the *Agent* oberves this next state S_{t+1}, and the cycle repeats (until we reach a terminal state). This cyclic interplay is depicted in Figure 1.1. Note that time ticks over from t to $t+1$ when the environment sees the state S_t and action A_t.

The MDP framework was formalized in a paper by Richard Bellman (Bellman 1957a) and the MDP theory was developed further in Richard Bellman's book named *Dynamic Programming* (Bellman 1957b) and in Ronald Howard's book named *Dynamic Programming and Markov Processes* (Howard 1960).

1.3 Formal Definition of a Markov Decision Process

Similar to the definitions of Markov Processes and Markov Reward Processes, for ease of exposition, the definitions and theory of Markov Decision Processes below will be for discrete-time, for countable state spaces and countable set of pairs of next state and reward transitions (with the knowledge that the definitions and theory are analogously extensible to continuous-time and uncountable spaces, which we shall indeed encounter later in this book).

Definition 1.3.1. A *Markov Decision Process* comprises of:

- A countable set of states $\mathcal{S}$ (known as the State Space), a set $\mathcal{T} \subseteq \mathcal{S}$ (known as the set of Terminal States), and a countable set of actions $\mathcal{A}$ (known as the Action Space).
- A time-indexed sequence of environment-generated random states $S_t \in \mathcal{S}$ for time

steps $t = 0, 1, 2, \ldots$, a time-indexed sequence of environment-generated *Reward* random variables $R_t \in \mathcal{D}$ (a countable subset of $\mathbb{R}$) for time steps $t = 1, 2, \ldots$, and a time-indexed sequence of agent-controllable actions $A_t \in \mathcal{A}$ for time steps $t = 0, 1, 2, \ldots$. (Sometimes we restrict the set of actions allowable from specific states, in which case, we abuse the $\mathcal{A}$ notation to refer to a function whose domain is $\mathcal{N}$ and range is $\mathcal{A}$, and we say that the set of actions allowable from a state $s \in \mathcal{N}$ is $\mathcal{A}(s)$.)

- Markov Property:

$$\mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0)] = \mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t)] \text{ for all} t \geq 0$$

- Termination: If an outcome for S_T (for some time step T) is a state in the set $\mathcal{T}$, then this sequence outcome terminates at time step T.

As in the case of Markov Reward Processes, we denote the set of non-terminal states $\mathcal{S} - \mathcal{T}$ as $\mathcal{N}$ and refer to any state in $\mathcal{N}$ as a non-terminal state. The sequence:

$$S_0, A_0, R_1, S_1, A_1, R_1, S_2, \ldots$$

terminates at time step T if $S_T \in \mathcal{T}$ (i.e., the final reward is R_T and the final action is A_{T-1}).

In the more general case, where states or rewards are uncountable, the same concepts apply except that the mathematical formalism needs to be more detailed and more careful. Specifically, we'd end up with integrals instead of summations, and probability density functions (for continuous probability distributions) instead of probability mass functions (for discrete probability distributions). For ease of notation and more importantly, for ease of understanding of the core concepts (without being distracted by heavy mathematical formalism), we've chosen to stay with discrete-time, countable $\mathcal{S}$, countable $\mathcal{A}$ and countable $\mathcal{D}$ (by default). However, there will be examples of Markov Decision Processes in this book involving continuous-time and uncountable $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{D}$ (please adjust the definitions and formulas accordingly).

We refer to $\mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t)]$ as the transition probabilities of the Markov Decision Process for time t.

As in the case of Markov Processes and Markov Reward Processes, we shall (by default) assume Time-Homogeneity for Markov Decision Processes, i.e., $\mathbb{P}[(R_{t+1}, S_{t+1})|(S_t, A_t)]$ is independent of t. This means the transition probabilities of a Markov Decision Process can, in the most general case, be expressed as a state-reward transition probability function:

$$\mathcal{P}_R : \mathcal{N} \times \mathcal{A} \times \mathcal{D} \times \mathcal{S} \to [0, 1]$$

defined as:

$$\mathcal{P}_R(s, a, r, s') = \mathbb{P}[(R_{t+1} = r, S_{t+1} = s')|(S_t = s, A_t = a)]$$

$$\text{for time steps } t = 0, 1, 2, \ldots, \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}, r \in \mathcal{D}, s' \in \mathcal{N} \text{ such that}$$

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{D}} \mathcal{P}_R(s, a, r, s') = 1 \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

Henceforth, any time we say Markov Decision Process, assume we are refering to a Discrete-Time, Time-Homogeneous Markov Decision Process with countable spaces and countable transitions (unless explicitly specified otherwise), which in turn can be characterized by the state-reward transition probability function $\mathcal{P}_R$. Given a specification of $\mathcal{P}_R$, we can construct:

- The state transition probability function

$$\mathcal{P} : \mathcal{N} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$$

defined as:

$$\mathcal{P}(s, a, s') = \sum_{r \in \mathcal{D}} \mathcal{P}_R(s, a, r, s')$$

- The reward transition function:

$$\mathcal{R}_T : \mathcal{N} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$$

defined as:

$$\mathcal{R}_T(s, a, s') = \mathbb{E}[R_{t+1} | (S_{t+1} = s', S_t = s, A_t = a)]$$

$$= \sum_{r \in \mathcal{D}} \frac{\mathcal{P}_R(s, a, r, s')}{\mathcal{P}(s, a, s')} \cdot r = \sum_{r \in \mathcal{D}} \frac{\mathcal{P}_R(s, a, r, s')}{\sum_{r \in \mathcal{D}} \mathcal{P}_R(s, a, r, s')} \cdot r$$

The Rewards specification of most Markov Decision Processes we encounter in practice can be directly expressed as the reward transition function $\mathcal{R}_T$ (versus the more general specification of $\mathcal{P}_R$). Lastly, we want to highlight that we can transform either of $\mathcal{P}_R$ or $\mathcal{R}_T$ into a "more compact" reward function that is sufficient to perform key calculations involving Markov Decision Processes. This reward function

$$\mathcal{R} : \mathcal{N} \times \mathcal{A} \to \mathbb{R}$$

is defined as:

$$\mathcal{R}(s, a) = \mathbb{E}[R_{t+1} | (S_t = s, A_t = a)]$$

$$= \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') \cdot \mathcal{R}_T(s, a, s') = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{D}} \mathcal{P}_R(s, a, r, s') \cdot r$$

1.4 Policy

Having understood the dynamics of a Markov Decision Process, we now move on to the specification of the *Agent*'s actions as a function of the current state. In the general case, we assume that the Agent will perform a random action A_t, according to a probability distribution that is a function of the current state S_t. We refer to this function as a *Policy*. Formally, a *Policy* is a function

$$\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$$

defined as:

$$\pi(s, a) = \mathbb{P}[A_t = a | S_t = s] \text{ for time steps } t = 0, 1, 2, \ldots, \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

such that

$$\sum_{a \in \mathcal{A}} \pi(s, a) = 1 \text{ for all } s \in \mathcal{N}$$

Note that the definition above assumes that a Policy is Markovian, i.e., the action probabilities depend only on the current state and not the history. The definition above also assumes that a Policy is *Stationary*, i.e., $\mathbb{P}[A_t = a | S_t = s]$ is invariant in time t. If we do encounter a situation where the policy would need to depend on the time t, we'll simply include t to be part of the state, which would make the Policy stationary (albeit at the cost of state-space bloat and hence, computational cost).

When we have a policy such that the action probability distribution for each state is concentrated on a single action, we refer to it as a deterministic policy. Formally, a deterministic policy $\pi_D : \mathcal{N} \to \mathcal{A}$ has the property that for all $s \in \mathcal{N}$,

$$\pi(s, \pi_D(s)) = 1 \text{ and } \pi(s, a) = 0 \text{ for all } a \in \mathcal{A} \text{ with } a \neq \pi_D(s)$$

So we shall denote deterministic policies simply as the function π_D. We shall refer to non-deterministic policies as stochastic policies (the word stochastic reflecting the fact that the agent will perform a random action according to the probability distribution specified by π). So when we use the notation π, assume that we are dealing with a stochastic (i.e., non-deterministic) policy and when we use the notation π_D, assume that we are dealing with a deterministic policy.

Let's write some code to get a grip on the concept of Policy. We start with the design of an abstract class called `Policy` that represents a general Policy, as we have articulated above. The only method it contains is an abstract method `act` that accepts as input a `state: NonTerminal[S]` (as seen before in the classes `MarkovProcess` and `MarkovRewardProcess`, `S` is a generic type to represent a generic state space) and produces as output a `Distribution[A]` representing the probability distribution of the random action as a function of the input non-terminal state. Note that `A` represents a generic type to represent a generic action space.

```
A = TypeVar('A')
S = TypeVar('S')

class Policy(ABC, Generic[S, A]):

    @abstractmethod
    def act(self, state: NonTerminal[S]) -> Distribution[A]:
        pass
```

Next, we implement a class for deterministic policies.

```
@dataclass(frozen=True)
class DeterministicPolicy(Policy[S, A]):
    action_for: Callable[[S], A]

    def act(self, state: NonTerminal[S]) -> Constant[A]:
        return Constant(self.action_for(state.state))
```

We will often encounter policies that assign equal probabilities to all actions, from each non-terminal state. We implement this class of policies as follows:

```
from rl.distribution import Choose

@dataclass(frozen=True)
class UniformPolicy(Policy[S, A]):
    valid_actions: Callable[[S], Iterable[A]]

    def act(self, state: NonTerminal[S]) -> Choose[A]:
        return Choose(self.valid_actions(state.state))
```

The above code is in the file rl/policy.py.

Now let's write some code to create some concrete policies for an example we are familiar with - the simple inventory example. We first create a concrete class `SimpleInventoryDeterministicPolicy` for deterministic inventory replenishment policies that is a derived class of `DeterministicPolicy`. Note that the generic state type `S` is replaced here with the class `InventoryState` that represents a state in the inventory example, comprising of the On-Hand and On-Order inventory quantities. Also note that the generic action type `A` is replaced here with the `int` type since in this example, the action is the quantity of inventory to be ordered at store-closing (which is an integer quantity). Invoking the `act` method of `SimpleInventoryDeterministicPolicy` runs the following deterministic policy:

$$\pi_D((\alpha, \beta)) = \max(r - (\alpha + \beta), 0)$$

where r is a parameter representing the "reorder point" (meaning, we order only when the inventory position falls below the "reorder point"), α is the On-Hand Inventory at store-closing, β is the On-Order Inventory at store-closing, and inventory position is equal to $\alpha + \beta$. In Chapter **??**, we set the reorder point to be equal to the store capacity C.

```
from rl.distribution import Constant

@dataclass(frozen=True)
class InventoryState:
    on_hand: int
    on_order: int

    def inventory_position(self) -> int:
        return self.on_hand + self.on_order

class SimpleInventoryDeterministicPolicy(
        DeterministicPolicy[InventoryState, int]
):
    def __init__(self, reorder_point: int):
        self.reorder_point: int = reorder_point

        def action_for(s: InventoryState) -> int:
            return max(self.reorder_point - s.inventory_position(), 0)

        super().__init__(action_for)
```

We can instantiate a specific deterministic policy with a reorder point of say 8 as:

```
si_dp = SimpleInventoryDeterministicPolicy(reorder_point=8)
```

Now let's write some code to create stochastic policies for the inventory example. We create a concrete class `SimpleInventoryStochasticPolicy` that implements the interface of the abstract class `Policy` (specifically implements the abstract method `act`). The code in `act` implements a stochastic policy as a `SampledDistribution[int]` driven by a sampling of the Poison distribution for the reorder point. Specifically, the reorder point r is treated as a Poisson random variable with a specified mean (of say $\lambda \in \mathbb{R}_{\geq 0}$). We sample a value of the reorder point r from this Poisson distribution (with mean λ). Then, we create a sample order quantity (*action*) $\theta \in \mathbb{Z}_{\geq 0}$ defined as:

$$\theta = \max(r - (\alpha + \beta), 0)$$

```
import numpy as np
from rl.distribution import SampledDistribution

class SimpleInventoryStochasticPolicy(Policy[InventoryState, int]):
    def __init__(self, reorder_point_poisson_mean: float):
```

```
        self.reorder_point_poisson_mean: float = reorder_point_poisson_mean
    def act(self, state: NonTerminal[InventoryState]) -> \
            SampledDistribution[int]:
        def action_func(state=state) -> int:
            reorder_point_sample: int = \
                np.random.poisson(self.reorder_point_poisson_mean)
            return max(
                reorder_point_sample - state.state.inventory_position(),
                0
            )
        return SampledDistribution(action_func)
```

We can instantiate a specific stochastic policy with a reorder point poisson distribution mean of say 8.0 as:

```
si_sp = SimpleInventoryStochasticPolicy(reorder_point_poisson_mean=8.0)
```

We will revisit the simple inventory example in a bit after we cover the code for Markov Decision Processes, when we'll show how to simulate the Markov Decision Process for this simple inventory example, with the agent running a deterministic policy. But before we move on to the code design for Markov Decision Processes (to accompany the above implementation of Policies), we need to cover an important insight linking Markov Decision Processes, Policies and Markov Reward Processes.

1.5 [Markov Decision Process, Policy] := Markov Reward Process

This section has an important insight - that if we evaluate a Markov Decision Process (MDP) with a fixed policy π (in general, with a fixed stochastic policy π), we get the Markov Reward Process (MRP) that is *implied* by the combination of the MDP and the policy π. Let's clarify this with notational precision. But first we need to point out that we have some notation clashes between MDP and MRP. We used $\mathcal{P}_R$ to denote the transition probability function of the MRP as well as to denote the state-reward transition probability function of the MDP. We used $\mathcal{P}$ to denote the transition probability function of the Markov Process implicit in the MRP as well as to denote the state transition probability function of the MDP. We used $\mathcal{R}_T$ to denote the reward transition function of the MRP as well as to denote the reward transition function of the MDP. We used $\mathcal{R}$ to denote the reward function of the MRP as well as to denote the reward function of the MDP. We can resolve these notation clashes by noting the arguments to $\mathcal{P}_R, \mathcal{P}, \mathcal{R}_T$ and $\mathcal{R}$, but to be extra-clear, we'll put a superscript of π to each of the functions $\mathcal{P}_R, \mathcal{P}, \mathcal{R}_T$ and $\mathcal{R}$ of the π-implied MRP so as to distinguish between these functions for the MDP versus the π-implied MRP.

Let's say we are given a fixed policy π and an MDP specified by it's state-reward transition probability function $\mathcal{P}_R$. Then the transition probability function $\mathcal{P}_R^\pi$ of the MRP implied by the evaluation of the MDP with the policy π is defined as:

$$\mathcal{P}_R^\pi(s, r, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{P}_R(s, a, r, s')$$

Likewise,

$$\mathcal{P}^\pi(s, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{P}(s, a, s')$$

$$\mathcal{R}_T^\pi(s, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{R}_T(s, a, s')$$

$$\mathcal{R}^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{R}(s, a)$$

So any time we talk about an MDP evaluated with a fixed policy, you should know that we are effectively talking about the implied MRP. This insight is now going to be key in the design of our code to represent Markov Decision Processes.

We create an abstract class called `MarkovDecisionProcess` (code shown below) with two abstract methods - `step` and `actions`. The `step` method is key: it is meant to specify the distribution of pairs of next state and reward, given a non-terminal state and action. The `actions` method's interface specifies that it takes as input a `state: NonTerminal[S]` and produces as output an `Iterable[A]` to represent the set of actions allowable for the input `state` (since the set of actions can be potentially infinite - in which case we'd have to return an `Iterator[A]` - the return type is fairly generic, i.e., `Iterable[A]`).

The `apply_policy` method takes as input a `policy: Policy[S, A]` and returns a `MarkovRewardProcess` representing the implied MRP. Let's understand the code in `apply_policy`: First, we construct a `class RewardProcess` that implements the abstract method `transition_reward` of `MarkovRewardProcess`. `transition_reward` takes as input a `state: NonTerminal[S]`, creates `actions: Distribution[A]` by applying the given `policy` on `state`, and finally uses the `apply` method of `Distribution` to transform `actions: Distribution[A]` into a `Distribution[Tuple[State[S], float]]` (distribution of (next state, reward) pairs) using the abstract method `step`.

We also write the `simulate_actions` method that is analogous to the `simulate_reward` method we had written for `MarkovRewardProcess` for generating a sampling trace. In this case, each step in the sampling trace involves sampling an action from the given `policy` and then sampling the pair of next state and reward, given the `state` and sampled `action`. Each generated `TransitionStep` object consists of the 4-tuple: (state, action, next state, reward). Here's the actual code:

```
from rl.distribution import Distribution

@dataclass(frozen=True)
class TransitionStep(Generic[S, A]):
    state: NonTerminal[S]
    action: A
    next_state: State[S]
    reward: float

class MarkovDecisionProcess(ABC, Generic[S, A]):
    @abstractmethod
    def actions(self, state: NonTerminal[S]) -> Iterable[A]:
        pass

    @abstractmethod
    def step(
        self,
        state: NonTerminal[S],
        action: A
    ) -> Distribution[Tuple[State[S], float]]:
        pass

    def apply_policy(self, policy: Policy[S, A]) -> MarkovRewardProcess[S]:
        mdp = self

        class RewardProcess(MarkovRewardProcess[S]):
            def transition_reward(
                self,
```

```
                state: NonTerminal[S]
            ) -> Distribution[Tuple[State[S], float]]:
                actions: Distribution[A] = policy.act(state)
                return actions.apply(lambda a: mdp.step(state, a))

        return RewardProcess()

    def simulate_actions(
            self,
            start_states: Distribution[NonTerminal[S]],
            policy: Policy[S, A]
    ) -> Iterable[TransitionStep[S, A]]:
        state: State[S] = start_states.sample()

        while isinstance(state, NonTerminal):
            action_distribution = policy.act(state)

            action = action_distribution.sample()
            next_distribution = self.step(state, action)

            next_state, reward = next_distribution.sample()
            yield TransitionStep(state, action, next_state, reward)
            state = next_state
```

The above code is in the file rl/markov_decision_process.py.

1.6 Simple Inventory Example with Unlimited Capacity (Infinite State/Action Space)

Now we come back to our simple inventory example. Unlike previous situations of this example, here we assume that there is no space capacity constraint on toothpaste. This means we have a choice of ordering any (unlimited) non-negative integer quantity of toothpaste units. Therefore, the action space is infinite. Also, since the order quantity shows up as On-Order the next day and as delivered inventory the day after the next day, the On-Hand and On-Order quantities are also unbounded. Hence, the state space is infinite. Due to the infinite state and action spaces, we won't be able to take advantage of the so-called "Tabular Dynamic Programming Algorithms" we will cover in Chapter **??** (algorithms that are meant for finite state and action spaces). There is still significant value in modeling infinite MDPs of this type because we can perform simulations (by sampling from an infinite space). Simulations are valuable not just to explore various properties and metrics relevant in the real-world problem modeled with an MDP, but simulations also enable us to design approximate algorithms to calculate Value Functions for given policies as well as Optimal Value Functions (which is the ultimate purpose of modeling MDPs).

We will cover details on these approximate algorithms later in the book - for now, it's important for you to simply get familiar with how to model infinite MDPs of this type. This infinite-space inventory example serves as a great learning for an introduction to modeling an infinite (but countable) MDP.

We create a concrete class `SimpleInventoryMDPNoCap` that implements the abstract class `MarkovDecisionProcess` (specifically implements abstract methods `step` and `actions`). The attributes `poisson_lambda`, `holding_cost` and `stockout_cost` have the same semantics as what we had covered for Markov Reward Processes in Chapter **??** (`SimpleInventoryMRP`). The `step` method takes as input a `state: NonTerminal[InventoryState]` and an `order: int` (representing the MDP action). We sample from the poisson probability distribution of customer demand (calling it `demand_sample: int`). Using `order: int` and `demand_sample: int`, we obtain a sample of the pair of `next_state: InventoryState` and `reward: float`. This sample pair is returned as a `SampledDistribution` object. The above sampling dynamics

effectively describe the MDP in terms of this `step` method. The `actions` method returns an `Iterator[int]`, an infinite generator of non-negative integers to represent the fact that the action space (order quantities) for any state comprise of all non-negative integers.

```
import itertools
import numpy as np
from rl.distribution import SampledDistribution

@dataclass(frozen=True)
class SimpleInventoryMDPNoCap(MarkovDecisionProcess[InventoryState, int]):
    poisson_lambda: float
    holding_cost: float
    stockout_cost: float

    def step(
        self,
        state: NonTerminal[InventoryState],
        order: int
    ) -> SampledDistribution[Tuple[State[InventoryState], float]]:

        def sample_next_state_reward(
            state=state,
            order=order
        ) -> Tuple[State[InventoryState], float]:
            demand_sample: int = np.random.poisson(self.poisson_lambda)
            ip: int = state.state.inventory_position()
            next_state: InventoryState = InventoryState(
                max(ip - demand_sample, 0),
                order
            )
            reward: float = - self.holding_cost * state.state.on_hand\
                - self.stockout_cost * max(demand_sample - ip, 0)
            return NonTerminal(next_state), reward

        return SampledDistribution(sample_next_state_reward)

    def actions(self, state: NonTerminal[InventoryState]) -> Iterator[int]:
        return itertools.count(start=0, step=1)
```

We leave it to you as an exercise to run various simulations of the MRP implied by the deterministic and stochastic policy instances we had created earlier (the above code is in the file rl/chapter3/simple_inventory_mdp_nocap.py). See the method `fraction_of_days_oos` in this file as an example of a simulation to calculate the percentage of days when we'd be unable to satisfy some customer demand for toothpaste due to too little inventory at store-opening (naturally, the higher the re-order point in the policy, the lesser the percentage of days when we'd be Out-of-Stock). This kind of simulation exercise helps build intuition on the tradeoffs we have to make between having too little inventory versus having too much inventory (holding costs versus stockout costs) - essentially leading to our ultimate goal of determining the Optimal Policy (more on this later).

1.7 Finite Markov Decision Processes

Certain calculations for Markov Decision Processes can be performed easily if:

- The state space is finite ($\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$),
- The action space $\mathcal{A}(s)$ is finite for each $s \in \mathcal{N}$,
- The set of unique pairs of next state and reward transitions from each pair of current non-terminal state and action is finite.

If we satisfy the above three characteristics, we refer to the Markov Decision Process as a Finite Markov Decision Process. Let us write some code for a Finite Markov Decision Process. We create a concrete class `FiniteMarkovDecisionProcess` that implements the interface of the abstract class `MarkovDecisionProcess` (specifically implements the abstract methods `step` and the `actions`). Our first task is to think about the data structure required to specify an instance of `FiniteMarkovDecisionProcess` (i.e., the data structure we'd pass to the `__init__` method of `FiniteMarkovDecisionProcess`). Analogous to how we curried $\mathcal{P}_R$ for a Markov Reward Process as $\mathcal{N} \to (\mathcal{S} \times \mathcal{D} \to [0, 1])$ (where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and $\mathcal{N}$ has $m \leq n$ states), here we curry $\mathcal{P}_R$ for the MDP as:

$$\mathcal{N} \to (\mathcal{A} \to (\mathcal{S} \times \mathcal{D} \to [0, 1]))$$

Since $\mathcal{S}$ is finite, $\mathcal{A}$ is finite, and the set of next state and reward transitions for each pair of current state and action is also finite, we can represent $\mathcal{P}_R$ as a data structure of type `StateActionMapping[S, A]` as shown below:

```
StateReward = FiniteDistribution[Tuple[State[S], float]]
ActionMapping = Mapping[A, StateReward[S]]
StateActionMapping = Mapping[NonTerminal[S], ActionMapping[A, S]]
```

The constructor (`__init__` method) of `FiniteMarkovDecisionProcess` takes as input `mapping` which is essentially of the same structure as `StateActionMapping[S, A]`, except that the `Mapping` is specified in terms of `S` rather than `NonTerminal[S]` or `State[S]` so as to make it easy for a user to specify a `FiniteMarkovDecisionProcess` without the overhead of wrapping `S` in `NonTerminal[S]` or `Terminal[S]`. But this means `__init__` need to do the wrapping to construct the attribute `self.mapping: StateActionMapping[S, A]`. This represents the complete structure of the Finite MDP - it maps each non-terminal state to an action map, and it maps each action in each action map to a finite probability distribution of pairs of next state and reward (essentially the structure of the $\mathcal{P}_R$ function). Along with the attribute `self.mapping`, we also have an attribute `non_terminal_states: Sequence[NonTerminal[S]]` that is an ordered sequence of non-terminal states. Now let's consider the implementation of the abstract method `step` of `MarkovDecisionProcess`. It takes as input a `state: NonTerminal[S]` and an `action: A`. `self.mapping[state][action]` gives us an object of type `FiniteDistribution[Tuple[State[S], float]]` which represents a finite probability distribution of pairs of next state and reward, which is exactly what we want to return. This satisfies the responsibility of `FiniteMarkovDecisionProcess` in terms of implementing the abstract method `step` of the abstract class `MarkovDecisionProcess`. The other abstract method to implement is the `actions` method which produces an `Iterable` on the allowed actions $\mathcal{A}(s)$ for a given $s \in \mathcal{N}$ by invoking `self.mapping[state].keys()`. The `__repr__` method shown below is quite straightforward.

```
from rl.distribution import FiniteDistribution, SampledDistribution

class FiniteMarkovDecisionProcess(MarkovDecisionProcess[S, A]):

    mapping: StateActionMapping[S, A]
    non_terminal_states: Sequence[NonTerminal[S]]

    def __init__(
        self,
        mapping: Mapping[S, Mapping[A, FiniteDistribution[Tuple[S, float]]]]
    ):
        non_terminals: Set[S] = set(mapping.keys())
        self.mapping = {NonTerminal(s): {a: Categorical(
            {(NonTerminal(s1) if s1 in non_terminals else Terminal(s1), r): p
             for (s1, r), p in v}
```

```
        ) for a, v in d.items()} for s, d in mapping.items()}
        self.non_terminal_states = list(self.mapping.keys())

    def __repr__(self) -> str:
        display = ""
        for s, d in self.mapping.items():
            display += f"From State {s.state}:\n"
            for a, d1 in d.items():
                display += f"  With Action {a}:\n"
                for (s1, r), p in d1:
                    opt = "Terminal " if isinstance(s1, Terminal) else ""
                    display += f"    To [{opt}State {s1.state} and "\
                        + f"Reward {r:.3f}] with Probability {p:.3f}\n"
        return display

    def step(self, state: NonTerminal[S], action: A) -> StateReward[S]:
        action_map: ActionMapping[A, S] = self.mapping[state]
        return action_map[action]

    def actions(self, state: NonTerminal[S]) -> Iterable[A]:
        return self.mapping[state].keys()
```

Now that we've implemented a finite MDP, let's implement a finite policy that maps each non-terminal state to a probability distribution over a finite set of actions. So we create a concrete class `@dataclass FinitePolicy` that implements the interface of the abstract class `Policy` (specifically implements the abstract method `act`). An instance of `FinitePolicy` is specified with the attribute `self.policy_map: Mapping[S, FiniteDistribution[A]]` since this type captures the structure of the $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$ function in the curried form

$$\mathcal{N} \to (\mathcal{A} \to [0, 1])$$

for the case of finite $\mathcal{S}$ and finite $\mathcal{A}$. The `act` method is straightforward. We also implement a `__repr__` method for pretty-printing of `self.policy_map`.

```
@dataclass(frozen=True)
class FinitePolicy(Policy[S, A]):
    policy_map: Mapping[S, FiniteDistribution[A]]

    def __repr__(self) -> str:
        display = ""
        for s, d in self.policy_map.items():
            display += f"For State {s}:\n"
            for a, p in d:
                display += f"  Do Action {a} with Probability {p:.3f}\n"
        return display

    def act(self, state: NonTerminal[S]) -> FiniteDistribution[A]:
        return self.policy_map[state.state]
```

Let's also implement a finite deterministic policy as a derived class of `FinitePolicy`.

```
class FiniteDeterministicPolicy(FinitePolicy[S, A]):
    action_for: Mapping[S, A]

    def __init__(self, action_for: Mapping[S, A]):
        self.action_for = action_for
        super().__init__(policy_map={s: Constant(a) for s, a in
                                     self.action_for.items()})

    def __repr__(self) -> str:
        display = ""
        for s, a in self.action_for.items():
            display += f"For State {s}: Do Action {a}\n"
        return display
```

Armed with a `FinitePolicy` class, we can now write a method `apply_finite_policy` in `FiniteMarkovDecisionProcess` that takes as input a `policy: FinitePolicy[S, A]` and returns a `FiniteMarkovRewardProcess[S]` by processing the finite structures of both of the MDP and the Policy, and producing a finite structure of the implied MRP.

```
from collections import defaultdict
from rl.distribution import FiniteDistribution, Categorical

    def apply_finite_policy(self, policy: FinitePolicy[S, A])\
            -> FiniteMarkovRewardProcess[S]:

        transition_mapping: Dict[S, FiniteDistribution[Tuple[S, float]]] = {}

        for state in self.mapping:
            action_map: ActionMapping[A, S] = self.mapping[state]
            outcomes: DefaultDict[Tuple[S, float], float]\
                = defaultdict(float)
            actions = policy.act(state)
            for action, p_action in actions:
                for (s1, r), p in action_map[action]:
                    outcomes[(s1.state, r)] += p_action * p

            transition_mapping[state.state] = Categorical(outcomes)

        return FiniteMarkovRewardProcess(transition_mapping)
```

The code for `FiniteMarkovRewardProcess` is in rl/markov_decision_process.py and the code for `FinitePolicy` and `FiniteDeterministicPolicy` is in rl/policy.py.

1.8 Simple Inventory Example as a Finite Markov Decision Process

Now we'd like to model the simple inventory example as a Finite Markov Decision Process so we can take advantage of the algorithms specifically for Finite Markov Decision Processes. To enable finite states and finite actions, we now re-introduce the constraint of space capacity C and apply the restriction that the order quantity (action) cannot exceed $C - (\alpha + \beta)$ where α is the On-Hand component of the State and β is the On-Order component of the State. Thus, the action space for any given state $(\alpha, \beta) \in \mathcal{S}$ is finite. Next, note that this ordering policy ensures that in steady-state, the sum of On-Hand and On-Order will not exceed the capacity C. So we constrain the set of states to be the steady-state set of finite states

$$\mathcal{S} = \{(\alpha, \beta) | \alpha \in \mathbb{Z}_{\geq 0}, \beta \in \mathbb{Z}_{\geq 0}, 0 \leq \alpha + \beta \leq C\}$$

Although the set of states is finite, there are an infinite number of pairs of next state and reward outcomes possible from any given pair of current state and action. This is because there are an infinite set of possibilities of customer demand on any given day (resulting in infinite possibilities of stockout cost, i.e., negative reward, on any day). To qualify as a Finite Markov Decision Process, we need to model in a manner such that we have a finite set of pairs of next state and reward outcomes from any given pair of current state and action. So what we do is that instead of considering (S_{t+1}, R_{t+1}) as the pair of next state and reward, we model the pair of next state and reward to instead be $(S_{t+1}, \mathbb{E}[R_{t+1}|(S_t, S_{t+1}, A_t)])$ (we know $\mathcal{P}_R$ due to the Poisson probabilities of customer demand, so we can actually calculate this conditional expectation of reward). So given a state s and action a, the pairs of next state and reward would be: $(s', \mathcal{R}_T(s, a, s'))$ for all the s' we transition to from (s, a). Since the set of possible next states s' are finite, these newly-modeled rewards associated with the transitions $(\mathcal{R}_T(s, a, s'))$ are also finite and hence, the set of pairs of next state

and reward from any pair of current state and action are also finite. Note that this creative alteration of the reward definition is purely to reduce this Markov Decision Process into a Finite Markov Decision Process. Let's now work out the calculation of the reward transition function $\mathcal{R}_T$.

When the next state's (S_{t+1}) On-Hand is greater than zero, it means all of the day's demand was satisfied with inventory that was available at store-opening ($= \alpha + \beta$), and hence, each of these next states S_{t+1} correspond to no stockout cost and only an overnight holding cost of $h\alpha$. Therefore, for all α, β (with $0 \leq \alpha + \beta \leq C$) and for all order quantity (action) θ (with $0 \leq \theta \leq C - (\alpha + \beta)$):

$$\mathcal{R}_T((\alpha, \beta), \theta, (\alpha + \beta - i, \theta)) = -h\alpha \text{ for } 0 \leq i \leq \alpha + \beta - 1$$

When next state's (S_{t+1}) On-Hand is equal to zero, there are two possibilities:

1. The demand for the day was exactly $\alpha + \beta$, meaning all demand was satisifed with available store inventory (so no stockout cost and only overnight holding cost), or
2. The demand for the day was strictly greater than $\alpha + \beta$, meaning there's some stockout cost in addition to overnight holding cost. The exact stockout cost is an expectation calculation involving the number of units of missed demand under the corresponding poisson probabilities of demand exceeding $\alpha + \beta$.

This calculation is shown below:

$$\mathcal{R}_T((\alpha, \beta), \theta, (0, \theta)) = -h\alpha - p(\sum_{j=\alpha+\beta+1}^{\infty} f(j) \cdot (j - (\alpha + \beta)))$$

$$= -h\alpha - p(\lambda(1 - F(\alpha + \beta - 1)) - (\alpha + \beta)(1 - F(\alpha + \beta)))$$

So now we have a specification of $\mathcal{R}_T$, but when it comes to our coding interface, we are expected to specify $\mathcal{P}_R$ as that is the interface through which we create a `FiniteMarkovDecisionProcess`. Fear not - a specification of $\mathcal{P}_R$ is easy once we have a specification of $\mathcal{R}_T$. We simply create 5-tuples (s, a, r, s', p) for all $s \in \mathcal{N}, s' \in \mathcal{S}, a \in \mathcal{A}$ such that $r = \mathcal{R}_T(s, a, s')$ and $p = \mathcal{P}(s, a, s')$ (we know $\mathcal{P}$ along with $\mathcal{R}_T$), and the set of all these 5-tuples (for all $s \in \mathcal{N}, s' \in \mathcal{S}, a \in \mathcal{A}$) constitute the specification of $\mathcal{P}_R$, i.e., $\mathcal{P}_R(s, a, r, s') = p$. This turns our reward-definition-altered mathematical model of a Finite Markov Decision Process into a programming model of the `FiniteMarkovDecisionProcess` class. This reward-definition-altered model enables us to gain from the fact that we can leverage the algorithms we'll be writing for Finite Markov Decision Processes (specifically, the classical Dynamic Programming algorithms - covered in Chapter **??**). The downside of this reward-definition-altered model is that it prevents us from generating sampling traces of the specific rewards encountered when transitioning from one state to another (because we no longer capture the probabilities of individual reward outcomes). Note that we can indeed perform simulations, but each transition step in the sampling trace will only show us the "mean reward" (specifically, the expected reward conditioned on current state, action and next state).

In fact, most Markov Processes you'd encounter in practice can be modeled as a combination of $\mathcal{R}_T$ and $\mathcal{P}$, and you'd simply follow the above $\mathcal{R}_T$ to $\mathcal{P}_R$ representation transformation drill to present this information in the form of $\mathcal{P}_R$ to instantiate a `FiniteMarkovDecisionProcess`. We designed the interface to accept $\mathcal{P}_R$ as input since that is the most general interface for specifying Markov Decision Processes.

So now let's write some code for the simple inventory example as a Finite Markov Decision Process as described above. All we have to do is to create a derived class inherited from `FiniteMarkovDecisionProcess` and write a method to construct the `mapping` (i.e., $\mathcal{P}_R$) that the `__init__` constuctor of `FiniteMarkovRewardProcess` requires as input. Note that the generic state type `S` is replaced here with the `@dataclass InventoryState` to represent the inventory state, comprising of the On-Hand and On-Order inventory quantities, and the generic action type `A` is replaced here with `int` to represent the order quantity.

```
from scipy.stats import poisson
from rl.distribution import Categorical

InvOrderMapping = Mapping[
    InventoryState,
    Mapping[int, Categorical[Tuple[InventoryState, float]]]
]

class SimpleInventoryMDPCap(FiniteMarkovDecisionProcess[InventoryState, int]):

    def __init__(
        self,
        capacity: int,
        poisson_lambda: float,
        holding_cost: float,
        stockout_cost: float
    ):
        self.capacity: int = capacity
        self.poisson_lambda: float = poisson_lambda
        self.holding_cost: float = holding_cost
        self.stockout_cost: float = stockout_cost

        self.poisson_distr = poisson(poisson_lambda)
        super().__init__(self.get_action_transition_reward_map())

    def get_action_transition_reward_map(self) -> InvOrderMapping:
        d: Dict[InventoryState, Dict[int, Categorical[Tuple[InventoryState,
                                                            float]]]] = {}

        for alpha in range(self.capacity + 1):
            for beta in range(self.capacity + 1 - alpha):
                state: InventoryState = InventoryState(alpha, beta)
                ip: int = state.inventory_position()
                base_reward: float = - self.holding_cost * alpha
                d1: Dict[int, Categorical[Tuple[InventoryState, float]]] = {}

                for order in range(self.capacity - ip + 1):
                    sr_probs_dict: Dict[Tuple[InventoryState, float], float] =\
                        {(InventoryState(ip - i, order), base_reward):
                         self.poisson_distr.pmf(i) for i in range(ip)}

                    probability: float = 1 - self.poisson_distr.cdf(ip - 1)
                    reward: float = base_reward - self.stockout_cost *\
                        (probability * (self.poisson_lambda - ip) +
                         ip * self.poisson_distr.pmf(ip))
                    sr_probs_dict[(InventoryState(0, order), reward)] = \
                        probability
                    d1[order] = Categorical(sr_probs_dict)

                d[state] = d1
        return d
```

Now let's test this out with some example inputs (as shown below). We construct an instance of the `SimpleInventoryMDPCap` class with these inputs (named `si_mdp` below), then construct an instance of the `FinitePolicy[InventoryState, int]` class (a deterministic policy, named `fdp` below), and combine them to produce the implied MRP (an instance of the `FiniteMarkovRewardProcess[InventoryState]` class).

```
user_capacity = 2
user_poisson_lambda = 1.0
user_holding_cost = 1.0
user_stockout_cost = 10.0

si_mdp: FiniteMarkovDecisionProcess[InventoryState, int] =\
    SimpleInventoryMDPCap(
        capacity=user_capacity,
        poisson_lambda=user_poisson_lambda,
        holding_cost=user_holding_cost,
        stockout_cost=user_stockout_cost
    )

fdp: FiniteDeterministicPolicy[InventoryState, int] = \
    FiniteDeterministicPolicy(
        {InventoryState(alpha, beta): user_capacity - (alpha + beta)
         for alpha in range(user_capacity + 1)
         for beta in range(user_capacity + 1 - alpha)}
)

implied_mrp: FiniteMarkovRewardProcess[InventoryState] =\
    si_mdp.apply_finite_policy(fdp)
```

The above code is in the file rl/chapter3/simple_inventory_mdp_cap.py. We encourage you to play with the inputs in `__main__`, produce the resultant implied MRP, and explore it's characteristics (such as it's Reward Function and it's Value Function).

1.9 MDP Value Function for a Fixed Policy

Now we are ready to talk about the Value Function for an MDP evaluated with a fixed policy π (also known as the MDP *Prediction* problem). The term *Prediction* refers to the fact that this problem is about forecasting the expected future returns when the agent follows a specific policy. Just like in the case of MRP, we define the Return G_t at time step t for an MDP as:

$$G_t = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} \cdot R_i = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots$$

where $\gamma \in [0, 1]$ is a specified discount factor.

We use the above definition of *Return* even for a terminating sequence (say terminating at $t = T$, i.e., $S_T \in \mathcal{T}$), by treating $R_i = 0$ for all $i > T$.

The Value Function for an MDP evaluated with a fixed policy π

$$V^\pi : \mathcal{N} \to \mathbb{R}$$

is defined as:

$$V^\pi(s) = \mathbb{E}_{\pi, \mathcal{P}_R}[G_t | S_t = s] \text{ for all } s \in \mathcal{N}, \text{ for all } t = 0, 1, 2, \ldots$$

For the rest of the book, we assume that whenever we are talking about a Value Function, the discount factor γ is appropriate to ensure that the Expected Return from each state is finite - in particular, $\gamma < 1$ for continuing (non-terminating) MDPs where the Return could otherwise diverge.

We expand $V^\pi(s) = \mathbb{E}_{\pi, \mathcal{P}_R}[G_t | S_t = s]$ as follows:

$$
\begin{aligned}
&\mathbb{E}_{\pi,\mathcal{P}_R}[R_{t+1}|S_t = s] + \gamma \cdot \mathbb{E}_{\pi,\mathcal{P}_R}[R_{t+2}|S_t = s] + \gamma^2 \cdot \mathbb{E}_{\pi,\mathcal{P}_R}[R_{t+3}|S_t = s] + \ldots \\
&= \sum_{a\in\mathcal{A}} \pi(s,a) \cdot \mathcal{R}(s,a) + \gamma \cdot \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{N}} \mathcal{P}(s,a,s') \sum_{a'\in\mathcal{A}} \pi(s',a') \cdot \mathcal{R}(s',a') \\
&\quad + \gamma^2 \cdot \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{N}} \mathcal{P}(s,a',s') \sum_{a'\in\mathcal{A}} \pi(s',a') \sum_{s''\in\mathcal{N}} \mathcal{P}(s',a'',s'') \sum_{a''\in\mathcal{A}} \pi(s'',a'') \cdot \mathcal{R}(s'',a'') \\
&\quad + \ldots \\
&= \mathcal{R}^\pi(s) + \gamma \cdot \sum_{s'\in\mathcal{N}} \mathcal{P}^\pi(s,s') \cdot \mathcal{R}^\pi(s') + \gamma^2 \cdot \sum_{s'\in\mathcal{N}} \mathcal{P}^\pi(s,s') \sum_{s''\in\mathcal{N}} \mathcal{P}^\pi(s',s'') \cdot \mathcal{R}^\pi(s'') + \ldots
\end{aligned}
$$

But from Equation (**??**) in Chapter **??**, we know that the last expression above is equal to the π-implied MRP's Value Function for state s. So, the Value Function V^π of an MDP evaluated with a fixed policy π is exactly the same function as the Value Function of the π-implied MRP. So we can apply the MRP Bellman Equation on V^π, i.e.,

$$
\begin{aligned}
V^\pi(s) &= \mathcal{R}^\pi(s) + \gamma \cdot \sum_{s'\in\mathcal{N}} \mathcal{P}^\pi(s,s') \cdot V^\pi(s') \\
&= \sum_{a\in\mathcal{A}} \pi(s,a) \cdot \mathcal{R}(s,a) + \gamma \cdot \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s') \\
&= \sum_{a\in\mathcal{A}} \pi(s,a) \cdot (\mathcal{R}(s,a) + \gamma \cdot \sum_{s'\in\mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s')) \text{ for all } s \in \mathcal{N}
\end{aligned} \tag{1.1}
$$

As we saw in Chapter **??**, for finite state spaces that are not too large, Equation (1.1) can be solved for V^π (i.e. solution to the MDP *Prediction* problem) with a linear algebra solution (Equation (**??**) from Chapter **??**). More generally, Equation (1.1) will be a key equation for the rest of the book in developing various Dynamic Programming and Reinforcement Algorithms for the MDP *Prediction* problem. However, there is another Value Function that's also going to be crucial in developing MDP algorithms - one which maps a (state, action) pair to the expected return originating from the (state, action) pair when evaluated with a fixed policy. This is known as the *Action-Value Function* of an MDP evaluated with a fixed policy π:

$$Q^\pi : \mathcal{N} \times \mathcal{A} \to \mathbb{R}$$

defined as:

$$Q^\pi(s,a) = \mathbb{E}_{\pi,\mathcal{P}_R}[G_t|(S_t = s, A_t = a)] \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}, \text{ for all } t = 0, 1, 2, \ldots$$

To avoid terminology confusion, we refer to V^π as the *State-Value Function* (albeit often simply abbreviated to *Value Function*) for policy π, to distinguish from the *Action-Value Function* Q^π. The way to interpret $Q^\pi(s,a)$ is that it's the Expected Return from a given non-terminal state s by first taking the action a and subsequently following policy π. With this interpretation of $Q^\pi(s,a)$, we can perceive $V^\pi(s)$ as the "weighted average" of $Q^\pi(s,a)$ (over all possible actions a from a non-terminal state s) with the weights equal to probabilities of action a, given state s (i.e., $\pi(s,a)$). Precisely,

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(s,a) \cdot Q^\pi(s,a) \text{ for all } s \in \mathcal{N} \tag{1.2}$$

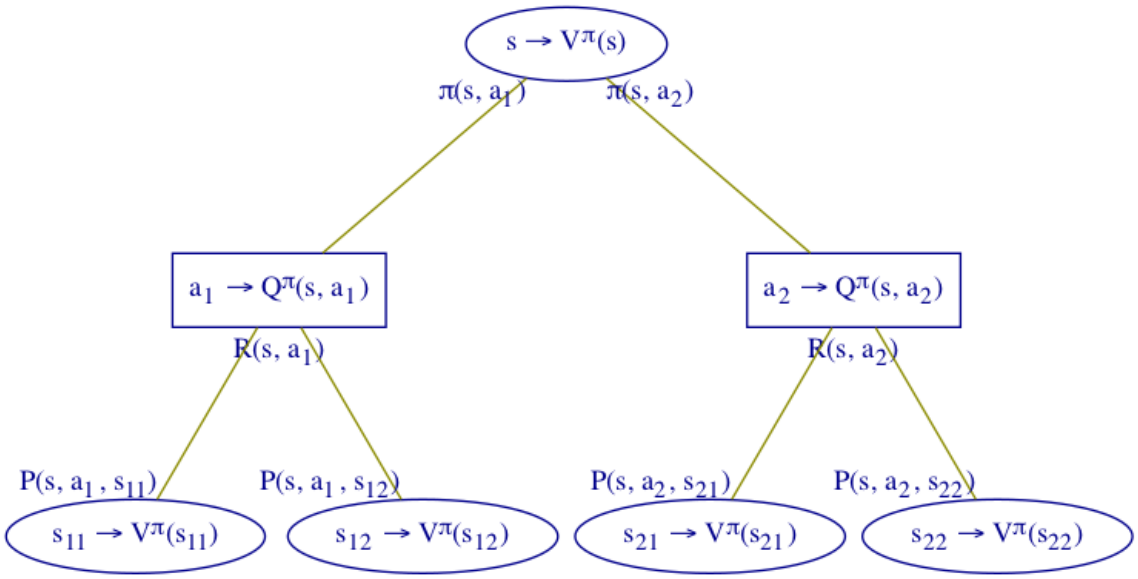

Figure 1.2: Visualization of MDP State-Value Function Bellman Policy Equation

Combining Equation (1.1) and Equation (1.2) yields:

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{1.3}$$

Combining Equation (1.3) and Equation (1.2) yields:

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \sum_{a' \in \mathcal{A}} \pi(s',a') \cdot Q^\pi(s',a') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{1.4}$$

Equation (1.1) is known as the MDP State-Value Function Bellman Policy Equation (Figure 1.2 serves as a visualization aid for this Equation). Equation (1.4) is known as the MDP Action-Value Function Bellman Policy Equation (Figure 1.3 serves as a visualization aid for this Equation). Note that Equation (1.2) and Equation (1.3) are embedded in Figure 1.2 as well as in Figure 1.3. Equations (1.1), (1.2), (1.3) and (1.4) are collectively known as the MDP Bellman Policy Equations.

For the rest of the book, in these MDP transition figures, we shall always depict states as elliptical-shaped nodes and actions as rectangular-shaped nodes. Notice that transition from a state node to an action node is associated with a probability represented by π and transition from an action node to a state node is associated with a probability represented by $\mathcal{P}$.

Note that for finite MDPs of state space not too large, we can solve the MDP Prediction problem (solving for V^π and equivalently, Q^π) in a straightforward manner: Given a policy π, we can create the finite MRP implied by π, using the method `apply_policy` in `FiniteMarkovDecisionProcess`, then use the direct linear-algebraic solution that we covered in Chapter **??** to calculate the Value Function of the π-implied MRP. We know that the π-implied MRP's Value Function is the same as the State-Value Function V^π of the MDP which can then be used to arrive at the Action-Value Function Q^π of the MDP (using Equation (1.3)). For large state spaces, we need to use iterative/numerical methods (Dynamic Programming and Reinforcement Learning algorithms) to solve this Prediction problem (covered later in this book).

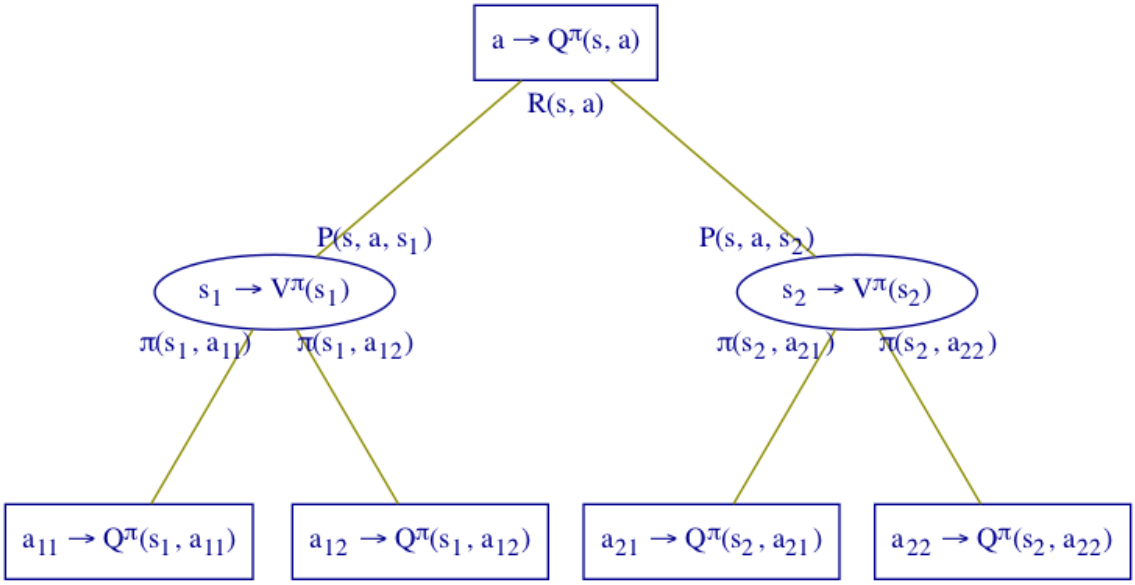

Figure 1.3: Visualization of MDP Action-Value Function Bellman Policy Equation

1.10 Optimal Value Function and Optimal Policies

Finally, we arrive at the main purpose of a Markov Decision Process - to identify a policy (or policies) that would yield the Optimal Value Function (i.e., the best possible *Expected Return* from each of the non-terminal states). We say that a Markov Decision Process is "solved" when we identify its Optimal Value Function (together with its associated Optimal Policy, i.e., a Policy that yields the Optimal Value Function). The problem of identifying the Optimal Value Function and its associated Optimal Policy/Policies is known as the MDP *Control* problem. The term *Control* refers to the fact that this problem involves steering the actions (by iterative modifications of the policy) to drive the Value Function towards Optimality. Formally, the Optimal Value Function

$$V^* : \mathcal{N} \to \mathbb{R}$$

is defined as:

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s) \text{ for all } s \in \mathcal{N}$$

where Π is the set of stationary (stochastic) policies over the spaces of $\mathcal{N}$ and $\mathcal{A}$.

The way to read the above definition is that for each non-terminal state s, we consider all possible stochastic stationary policies π, and maximize $V^\pi(s)$ across all these choices of π. Note that the maximization over choices of π is done separately for each s, so it's conceivable that different choices of π might maximize $V^\pi(s)$ for different $s \in \mathcal{N}$. Thus, from the above definition of V^*, we can't yet talk about the notion of "An Optimal Policy." So, for now, let's just focus on the notion of Optimal Value Function, as defined above. Note also that we haven't yet talked about how to achieve the above-defined maximization through an algorithm - we have simply *defined* the Optimal Value Function.

Likewise, the Optimal Action-Value Function

$$Q^* : \mathcal{N} \times \mathcal{A} \to \mathbb{R}$$

is defined as:

$$Q^*(s,a) = \max_{\pi \in \Pi} Q^\pi(s,a) \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

V^* is often refered to as the Optimal State-Value Function to distinguish it from the Optimal Action-Value Function Q^* (although, for succinctness, V^* is often also refered to as simply the Optimal Value Function). To be clear, if someone says, Optimal Value Function, by default, they'd be refering to the Optimal State-Value Function V^* (not Q^*).

Much like how the Value Function(s) for a fixed policy have a recursive formulation, Bellman noted (Bellman 1957b) that we can create a recursive formulation for the Optimal Value Function(s). Let us start by unraveling the Optimal State-Value Function $V^*(s)$ for a given non-terminal state s - we consider all possible actions $a \in \mathcal{A}$ we can take from state s, and pick the action a that yields the best Action-Value from thereon, i.e., the action a that yields the best $Q^*(s,a)$. Formally, this gives us the following equation:

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a) \text{ for all } s \in \mathcal{N} \tag{1.5}$$

Likewise, let's think about what it means to be optimal from a given non-terminal-state and action pair (s,a), i.e, let's unravel $Q^*(s,a)$. First, we get the immediate expected reward $\mathcal{R}(s,a)$. Next, we consider all possible random states $s' \in \mathcal{S}$ we can transition to, and from each of those states which are non-terminal states, we recursively act optimally. Formally, this gives us the following equation:

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^*(s') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{1.6}$$

Substituting for $Q^*(s,a)$ from Equation (1.6) in Equation (1.5) gives:

$$V^*(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^*(s')\} \text{ for all } s \in \mathcal{N} \tag{1.7}$$

Equation (1.7) is known as the MDP State-Value Function Bellman Optimality Equation and is depicted in Figure 1.4 as a visualization aid.

Substituting for $V^*(s)$ from Equation (1.5) in Equation (1.6) gives:

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \max_{a' \in \mathcal{A}} Q^*(s',a') \text{ for all } s \in \mathcal{N}, a \in \mathcal{A} \tag{1.8}$$

Equation (1.8) is known as the MDP Action-Value Function Bellman Optimality Equation and is depicted in Figure 1.5 as a visualization aid.

Note that Equation (1.5) and Equation (1.6) are embedded in Figure 1.4 as well as in Figure 1.5. Equations (1.7), (1.5), (1.6) and (1.8) are collectively known as the MDP Bellman Optimality Equations. We should highlight that when someone says MDP Bellman Equation or simply Bellman Equation, unless they explicit state otherwise, they'd be refering to the MDP Bellman Optimality Equations (and typically specifically the MDP State-Value Function Bellman Optimality Equation). This is because the MDP Bellman Optimality Equations address the ultimate purpose of Markov Decision Processes - to identify the Optimal Value Function and the associated policy/policies that achieve the Optimal Value Function (i.e., enabling us to solve the MDP *Control* problem).

Again, it pays to emphasize that the Bellman Optimality Equations don't directly give us a recipe to calculate the Optimal Value Function or the policy/policies that achieve

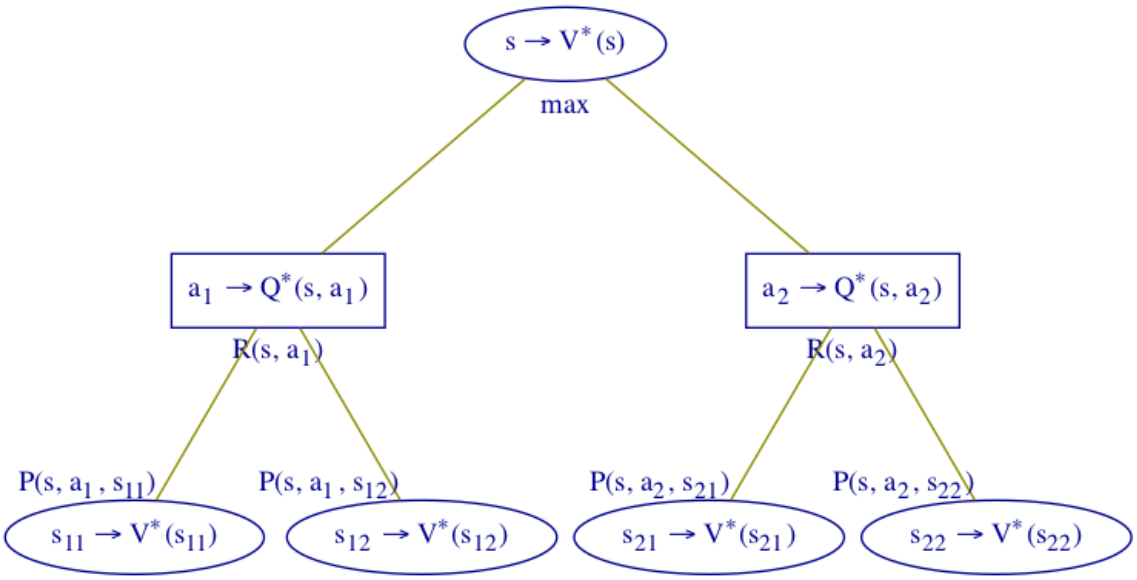

Figure 1.4: Visualization of MDP State-Value Function Bellman Optimality Equation

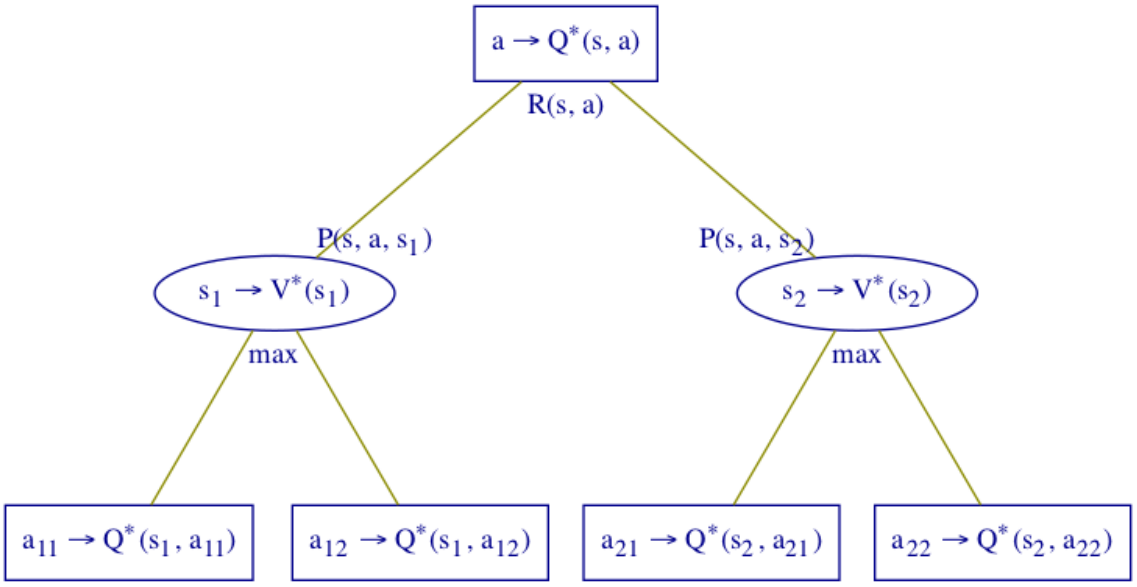

Figure 1.5: Visualization of MDP Action-Value Function Bellman Optimality Equation

the Optimal Value Function - they simply state a powerful mathematical property of the Optimal Value Function that (as we shall see later in this book) helps us come up with algorithms (Dynamic Programming and Reinforcement Learning) to calculate the Optimal Value Function and the associated policy/policies that achieve the Optimal Value Function.

We have been using the phrase "policy/policies that achieve the Optimal Value Function," but we haven't yet provided a clear definition of such a policy (or policies). In fact, as mentioned earlier, it's not clear from the definition of V^* if such a policy (one that would achieve V^*) exists (because it's conceivable that different policies π achieve the maximization of $V^\pi(s)$ for different states $s \in \mathcal{N}$). So instead, we define an *Optimal Policy* $\pi^* : \mathcal{N} \times \mathcal{A} \to [0, 1]$ as one that "dominates" all other policies with respect to the Value Functions for the policies. Formally,

$$\pi^* \in \Pi \text{ is an Optimal Policy if } V^{\pi^*}(s) \geq V^\pi(s) \text{ \textit{for all} } \pi \in \Pi \text{ and \textit{for all states} } s \in \mathcal{N}$$

The definition of an Optimal Policy π^* says that it is a policy that is "better than or equal to" (on the V^π metric) all other stationary policies *for all* non-terminal states (note that there could be multiple Optimal Policies). Putting this definition together with the definition of the Optimal Value Function V^*, the natural question to then ask is whether there exists an Optimal Policy π^* that maximizes $V^\pi(s)$ *for all* $s \in \mathcal{N}$, i.e., whether there exists a π^* such that $V^*(s) = V^{\pi^*}(s)$ for all $s \in \mathcal{N}$. On the face of it, this seems like a strong statement. However, this answers in the affirmative in most MDP settings of interest. The following theorem and proof is for our default setting of MDP (discrete-time, countable-spaces, time-homogeneous), but the statements and argument themes below apply to various other MDP settings as well. The MDP book by Martin Puterman (Puterman 2014) provides rigorous proofs for a variety of settings.

Theorem 1.10.1. *For any (discrete-time, countable-spaces, time-homogeneous) MDP:*

- *There exists an Optimal Policy $\pi^* \in \Pi$, i.e., there exists a Policy $\pi^* \in \Pi$ such that $V^{\pi^*}(s) \geq V^\pi(s)$ for all policies $\pi \in \Pi$ and for all states $s \in \mathcal{N}$*
- *All Optimal Policies achieve the Optimal Value Function, i.e. $V^{\pi^*}(s) = V^*(s)$ for all $s \in \mathcal{N}$, for all Optimal Policies π^**
- *All Optimal Policies achieve the Optimal Action-Value Function, i.e. $Q^{\pi^*}(s, a) = Q^*(s, a)$ for all $s \in \mathcal{N}$, for all $a \in \mathcal{A}$, for all Optimal Policies π^**

Before proceeding with the proof of Theorem (1.10.1), we establish a simple Lemma.

Lemma 1.10.2. *For any two Optimal Policies π_1^* and π_2^*, $V^{\pi_1^*}(s) = V^{\pi_2^*}(s)$ for all $s \in \mathcal{N}$*

Proof. Since π_1^* is an Optimal Policy, from the Optimal Policy definition, we have: $V^{\pi_1^*}(s) \geq V^{\pi_2^*}(s)$ for all $s \in \mathcal{N}$. Likewise, since π_2^* is an Optimal Policy, from the Optimal Policy definition, we have: $V^{\pi_2^*}(s) \geq V^{\pi_1^*}(s)$ for all $s \in \mathcal{N}$. This implies: $V^{\pi_1^*}(s) = V^{\pi_2^*}(s)$ for all $s \in \mathcal{N}$. □

Now we are ready to prove Theorem (1.10.1)

Proof. As a consequence of the above Lemma, all we need to do to prove Theorem (1.10.1) is to establish an Optimal Policy that achieves the Optimal Value Function and the Optimal Action-Value Function. We construct a Deterministic Policy (as a candidate Optimal Policy) $\pi^*_D : \mathcal{N} \to \mathcal{A}$ as follows:

$$\pi^*_D(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a) \text{ for all } s \in \mathcal{N} \tag{1.9}$$

Note that for any specific s, if two or more actions a achieve the maximization of $Q^*(s,a)$, then we use an arbitrary rule in breaking ties and assigning a single action a as the output of the above arg max operation.

First we show that π^*_D achieves the Optimal Value Functions V^* and Q^*. Since $\pi^*_D(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a)$ and $V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a)$ for all $s \in \mathcal{N}$, we can infer for all $s \in \mathcal{N}$ that:

$$V^*(s) = Q^*(s, \pi^*_D(s))$$

This says that we achieve the Optimal Value Function from a given non-terminal state s if we first take the action prescribed by the policy π^*_D (i.e., the action $\pi^*_D(s)$), followed by achieving the Optimal Value Function from each of the next time step's states. But note that each of the next time step's states can achieve the Optimal Value Function by doing the same thing described above ("first take action prescribed by π^*_D, followed by ..."), and so on and so forth for further time step's states. Thus, the Optimal Value Function V^* is achieved if from each non-terminal state, we take the action prescribed by π^*_D. Likewise, the Optimal Action-Value Function Q^* is achieved if from each non-terminal state, we take the action a (argument to Q^*) followed by future actions prescribed by π^*_D. Formally, this says:

$$V^{\pi^*_D}(s) = V^*(s) \text{ for all } s \in \mathcal{N}$$

$$Q^{\pi^*_D}(s,a) = Q^*(s,a) \text{ for all } s \in \mathcal{N}, \text{ for all } a \in \mathcal{A}$$

Finally, we argue that π^*_D is an Optimal Policy. Assume the contradiction (that π^*_D is not an Optimal Policy). Then there exists a policy $\pi \in \Pi$ and a state $s \in \mathcal{N}$ such that $V^\pi(s) > V^{\pi^*_D}(s)$. Since $V^{\pi^*_D}(s) = V^*(s)$, we have: $V^\pi(s) > V^*(s)$ which contradicts the Optimal Value Function Definition: $V^*(s) = \max_{\pi \in \Pi} V^\pi(s)$ for all $s \in \mathcal{N}$. Hence, π^*_D must be an Optimal Policy. □

Equation (1.9) is a key construction that goes hand-in-hand with the Bellman Optimality Equations in designing the various Dynamic Programming and Reinforcement Learning algorithms to solve the MDP Control problem (i.e., to solve for V^*, Q^* and π^*). Lastly, it's important to note that unlike the Prediction problem which has a straightforward linear-algebra-solver for small state spaces, the Control problem is non-linear and so, doesn't have an analogous straightforward linear-algebra-solver. The simplest solutions for the Control problem (even for small state spaces) are the Dynamic Programming algorithms we will cover in Chapter **??**.

1.11 Variants and Extensions of MDPs

1.11.1 Size of Spaces and Discrete versus Continuous

Variants of MDPs can be organized by variations in the size and type of:

- State Space
- Action Space
- Time Steps

State Space:

The definitions we've provided for MRPs and MDPs were for countable (discrete) state spaces. As a special case, we considered finite state spaces since we have pretty straightforward algorithms for exact solution of Prediction and Control problems for finite MDPs (which we shall learn about in Chapter **??**). We emphasize finite MDPs because they help you develop a sound understanding of the core concepts and make it easy to program the algorithms (known as "tabular" algorithms since we can represent the MDP in a "table," more specifically a Python data structure like `dict` or `numpy array`). However, these algorithms are practical only if the finite state space is not too large. Unfortunately, in many real-world problems, state spaces are either very large-finite or infinite (sometimes continuous-valued spaces). Large state spaces are unavoidable because phenomena in nature and metrics in business evolve in time due to a complex set of factors and often depend on history. To capture all these factors and to enable the Markov Property, we invariably end up with having to model large state spaces which suffer from two "curses":

- Curse of Dimensionality (size of state space $\mathcal{S}$)
- Curse of Modeling (size/complexity of state-reward transition probabilites $\mathcal{P}_R$)

Curse of Dimensionality is a term coined by Richard Bellman in the context of Dynamic Programming. It refers to the fact that when the number of dimensions in the state space grows, there is an exponential increase in the number of samples required to attain an adequate level of accuracy in algorithms. Consider this simple example (adaptation of an example by Bellman himself) - In a single dimension of space from 0 to 1, 100 evenly spaced sample points suffice to sample the space within a threshold distance of 0.01 between points. An equivalent sampling in 10 dimensions ($[0, 1]^{10}$) within a threshold distance of 0.01 between points will require 10^{20} points. So the 10-dimensional space requires points that are greater by a factor of 10^{18} relative to the points required in single dimension. This explosion in requisite points in the state space is known as the Curse of Dimensionality.

Curse of Modeling refers to the fact that when state spaces are large or when the structure of state-reward transition probabilities is complex, explicit modeling of these transition probabilities is very hard and often impossible (the set of probabilities can go beyond memory or even disk storage space). Even if it's possible to fit the probabilities in available storage space, estimating the actual probability values can be very difficult in complex real-world situations.

To overcome these two curses, we can attempt to contain the state space size with some dimensionality reduction techniques, i.e., including only the most relevant factors in the state representation. Secondly, if future outcomes depend on history, we can include just the past few time steps' values rather than the entire history in the state representation. These savings in state space size are essentially prudent approximations in the state representation. Such state space modeling considerations often require a sound understanding of the real-world problem. Recent advances in unsupervised Machine Learning can also help us contain the state space size. We won't discuss these modeling aspects in detail here - rather, we'd just like to emphasize for now that modeling the state space appropriately is one of the most important skills in real-world Reinforcement Learning, and we will illustrate some of these modeling aspects through a few examples later in this book.

Even after performing these modeling exercises in reducing the state space size, we often still end up with fairly large state spaces (so as to capture sufficient nuances of the real-world problem). We battle these two curses in fundamentally two (complementary) ways:

- Approximation of the Value Function - We create an approximate representation of the Value Function (eg: by using a supervised learning representation such as a neural network). This permits us to work with an appropriately sampled subset of the state space, infer the Value Function in this state space subset, and interpolate/extrapolate/generalize the Value Function in the remainder of the State Space.
- Sampling from the state-reward transition probabilities $\mathcal{P}_R$ - Instead of working with the explicit transition probabilities, we simply use the state-reward sample transitions and employ Reinforcement Learning algorithms to incrementally improve the estimates of the (approximated) Value Function. When state spaces are large, representing explicit transition probabilities is impossible (not enough storage space), and simply sampling from these probability distributions is our only option (and as you shall learn, is surprisingly effective).

This combination of sampling a state space subset, approximation of the Value Function (with deep neural networks), sampling state-reward transitions, and clever Reinforcement Learning algorithms goes a long way in breaking both the curse of dimensionality and curse of modeling. In fact, this combination is a common pattern in the broader field of Applied Mathematics to break these curses. The combination of Sampling and Function Approximation (particularly with the modern advances in Deep Learning) are likely to pave the way for future advances in the broader fields of Real-World AI and Applied Mathematics in general. We recognize that some of this discussion is a bit premature since we haven't even started teaching Reinforcement Learning yet. But we hope that this section provides some high-level perspective and connects the learnings from this chapter to the techniques/algorithms that will come later in this book. We will also remind you of this joint-importance of sampling and function approximation once we get started with Reinforcement Learning algorithms later in this book.

Action Space:

Similar to state spaces, the definitions we've provided for MDPs were for countable (discrete) action spaces. As a special case, we considered finite action spaces (together with finite state spaces) since we have pretty straightforward algorithms for exact solution of Prediction and Control problems for finite MDPs. As mentioned above, in these algorithms, we represent the MDP in Python data structures like `dict` or `numpy array`. However, these finite-MDP algorithms are practical only if the state and action spaces are not too large. In many real-world problems, action spaces do end up as fairly large - either finite-large or infinite (sometimes continuous-valued action spaces). The large size of the action space affects algorithms for MDPs in a couple of ways:

- Large action space makes the representation, estimation and evaluation of the policy π, of the Action-Value function for a policy Q^π and of the Optimal Action-Value function Q^* difficult. We have to resort to function approximation and sampling as ways to overcome the large size of the action space.
- The Bellman Optimality Equation leads to a crucial calculation step in Dynamic Programming and Reinforcement Learning algorithms that involves identifying the action for each non-terminal state that maximizes the Action-Value Function Q. When the action space is large, we cannot afford to evaluate Q for each action for an encountered state (as is done in simple tabular algorithms). Rather, we need to tap into an optimization algorithm to perform the maximization of Q over the action space, for each encountered state. Separately, there is a special class of Reinforcement Learning

algorithms called Policy Gradient Algorithms (that we shall later learn about) that are particularly valuable for large action spaces (where other types of Reinforcement Learning algorithms are not efficient and often, simply not an option). However, these techniques to deal with large action spaces require care and attention as they have their own drawbacks (more on this later).

Time Steps:

The definitions we've provided for MRP and MDP were for discrete time steps. We distinguish discrete time steps as terminating time-steps (known as terminating or episodic MRPs/MDPs) or non-terminating time-steps (known as continuing MRPs/MDPs). We've talked about how the choice of γ matters in these cases ($\gamma = 1$ doesn't work for some continuing MDPs because reward accumulation can blow up to infinity). We won't cover it in this book, but there is an alternative formulation of the Value Function as expected average reward (instead of expected discounted accumulated reward) where we don't discount even for continuing MDPs. We had also mentioned earlier that an alternative to discrete time steps is continuous time steps, which is convenient for analytical tractability.

Sometimes, even if state space and action space components have discrete values (eg: price of a security traded in fine discrete units, or number of shares of a security bought/sold on a given day), for modeling purposes, we sometimes find it convenient to represent these components as continuous values (i.e., uncountable state space). The advantage of continuous state/action space representation (especially when paired with continuous time) is that we get considerable mathematical benefits from differential calculus as well as from properties of continuous probability distributions (eg: gaussian distribution conveniences). In fact, continuous state/action space and continuous time are very popular in Mathematical Finance since some of the groundbreaking work from Mathematical Economics from the 1960s and 1970s - Robert Merton's Portfolio Optimization formulation and solution (Merton 1969) and Black-Scholes' Options Pricing model (Black and Scholes 1973), to name a couple - are grounded in stochastic calculus[1] which models stock prices/portfolio value as gaussian evolutions in continuous time (more on this later in the book) and treats trades (buy/sell quantities) as also continuous variables (permitting partial derivatives and tractable partial differential equations).

When all three of state space, action space and time steps are modeled as continuous, the Bellman Optimality Equation we covered in this chapter for countable spaces and discrete-time morphs into a differential calculus formulation and is known as the famous *Hamilton-Jacobi-Bellman* (HJB) equation[2]. The HJB Equation is commonly used to model and solve many problems in engineering, physics, economics and finance. We shall cover a couple of financial applications in this book that have elegant formulations in terms of the HJB equation and equally elegant analytical solutions of the Optimal Value Function and Optimal Policy (tapping into stochastic calculus and differential equations).

1.11.2 Partially-Observable Markov Decision Processes (POMDPs)

You might have noticed in the definition of MDP that there are actually two different notions of state, which we collapsed into a single notion of state. These two notions of state are:

[1] Appendix **??** provides a quick introduction to and overview of Stochastic Calculus.
[2] Appendix **??** provides a quick introduction to the HJB Equation.

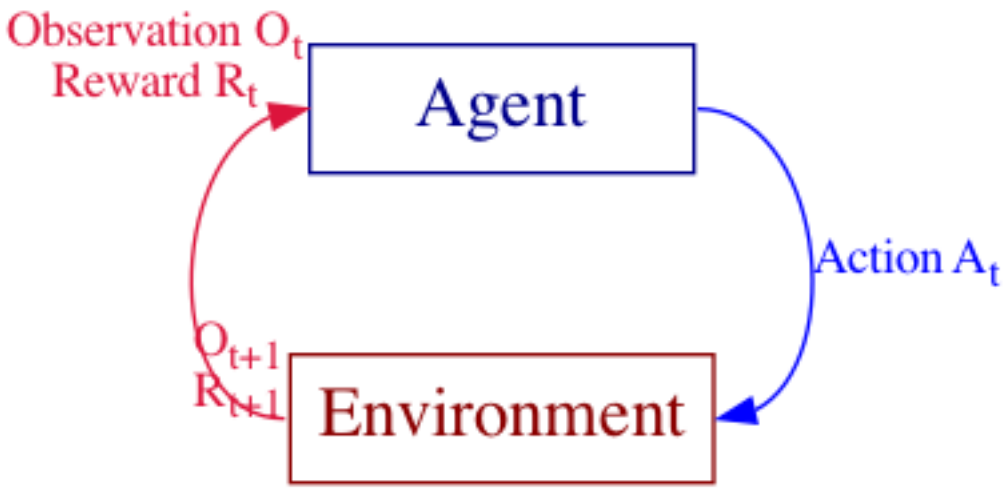

Figure 1.6: Partially-Observable Markov Decision Process

- The internal representation of the environment at each time step t (let's call it $S_t^{(e)}$). This internal representation of the environment is what drives the probabilistic transition to the next time step $t+1$, producing the random pair of next (environment) state $S_{t+1}^{(e)}$ and reward R_{t+1}.
- The agent state at each time step t (let's call it $S_t^{(a)}$). The agent state is what controls the action A_t the agent takes at time step t, i.e., the agent runs a policy π which is a function of the agent state $S_t^{(a)}$, producing a probability distribution of actions A_t.

In our definition of MDP, note that we implicitly assumed that $S_t^{(e)} = S_t^{(a)}$ at each time step t, and called it the (common) state S_t at time t. Secondly, we assumed that this state S_t is *fully observable* by the agent. To understand *full observability*, let us (first intuitively) understand the concept of *partial observability* in a more generic setting than what we had assumed in the framework for MDP. In this more generic framework, we denote O_t as the information available to the agent from the environment at time step t, as depicted in Figure 1.6. The notion of *partial observability* in this more generic framework is that from the history of observations, actions and rewards up to time step t, the agent does not have full knowledge of the environment state $S_t^{(e)}$. This lack of full knowledge of $S_t^{(e)}$ is known as *partial observability*. *Full observability*, on the other hand, means that the agent can fully construct $S_t^{(e)}$ as a function of the history of observations, actions and rewards up to time step t. Since we have the flexibility to model the exact data structures to represent observations, state and actions in this more generic framework, existence of full observability lets us re-structure the observation data at time step t to be $O_t = S_t^{(e)}$. Since we have also assumed $S_t^{(e)} = S_t^{(a)}$, we have:

$$O_t = S_t^{(e)} = S_t^{(a)} \text{ for all time steps } t = 0, 1, 2, \ldots$$

The above statement specialized the framework to that of Markov Decision Processes, which we can now name more precisely as Fully-Observable Markov Decision Processes (when viewed from the lens of the more generic framework described above, that permits partial observability or full observability).

In practice, you will often find that the agent doesn't know the true internal representation ($S_t^{(e)}$) of the environment (i.e, partial observability). Think about what it would take to know what drives a stock price from time step t to $t+1$ - the agent would need to have access to pretty much every little detail of trading activity in the entire world, and more!. However, since the MDP framework is simple and convenient, and since we have tractable Dynamic Programming and Reinforcement Learning algorithms to solve MDPs, we often do pretend that $O_t = S_t^{(e)} = S_t^{(a)}$ and carry on with our business of solving the assumed/modeled MDP. Often, this assumption of $O_t = S_t^{(e)} = S_t^{(a)}$ turns out to be a reasonable approximate model of the real-world but there are indeed situations where this assumption is far-fetched. These are situations where we have access to too little information pertaining to the key aspects of the internal state representation ($S_t^{(e)}$) of the environment. It turns out that we have a formal framework for these situations - this framework is known as Partially-Observable Markov Decision Process (POMDP for short). By default, the acronym MDP will refer to a Fully-Observable Markov Decision Process (i.e. corresponding to the MDP definition we have given earlier in this chapter). So let's now define a POMDP.

A POMDP has the usual features of an MDP (discrete-time, countable states, countable actions, countable next state-reward transition probabilities, discount factor, plus assuming time-homogeneity), together with the notion of random observation O_t at each time step t (each observation O_t lies within the Observation Space $\mathcal{O}$) and observation probability function $\mathcal{Z} : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \to [0, 1]$ defined as:

$$\mathcal{Z}(s', a, o) = \mathbb{P}[O_{t+1} = o | (S_{t+1} = s', A_t = a)]$$

It pays to emphasize that although a POMDP works with the notion of a state S_t, the agent doesn't have knowledge of S_t. It only has knowledge of observation O_t because O_t is the extent of information made available from the environment. The agent will then need to essentially "guess" (probabilistically) what the state S_t might be at each time step t in order to take the action A_t. The agent's goal in a POMDP is the same as that for an MDP: to determine the Optimal Value Function and to identify an Optimal Policy (achieving the Optimal Value Function).

Just like we have the rich theory and algorithms for MDPs, we have the theory and algorithms for POMDPs. POMDP theory is founded on the notion of *belief states*. The informal notion of a belief state is that since the agent doesn't get to see the state S_t (it only sees the observations O_t) at each time step t, the agent needs to keep track of what it thinks the state S_t might be, i.e., it maintains a probability distribution of states S_t conditioned on history. Let's make this a bit more formal.

Let us refer to the history H_t known to the agent at time t as the sequence of data it has collected up to time t. Formally, this data sequence H_t is:

$$(O_0, A_0, R_1, O_1, A_1, R_2, \ldots, O_{t-1}, A_{t-1}, R_t, O_t)$$

A Belief State $b(h)_t$ at time t is a probability distribution over states, conditioned on the history h, i.e.,

$$b(h)_t = (\mathbb{P}[S_t = s_1 | H_t = h], \mathbb{P}[S_t = s_2 | H_t = h], \ldots)$$

such that $\sum_{s \in \mathcal{S}} b(h)_t(s) = 1$ for all histories h and for each $t = 0, 1, 2, \ldots$.

Since the history H_t satisfies the Markov Property, the belief state $b(h)_t$ satisfies the Markov Property. So we can reduce the POMDP to an MDP M with the set of belief states of the POMDP as the set of states of the MDP M. Note that even if the set of states of

the POMDP were finite, the set of states of the MDP M will be infinite (i.e. infinite belief states). We can see that this will almost always end up as a giant MDP M. So although this is useful for theoretical reasoning, practically solving this MDP M is often quite hard computationally. However, specialized techniques have been developed to solve POMDPs but as you might expect, their computational complexity is still quite high. So we end up with a choice when encountering a POMDP - either try to solve it with a POMDP algorithm (computationally inefficient but capturing the reality of the real-world problem) or try to approximate it as an MDP (pretending $O_t = S_t^{(e)} = S_t^{(a)}$) which will likely be computationally more efficient but might be a gross approximation of the real-world problem, which in turn means it's effectiveness in practice might be compromised. This is the modeling dilemma we often end up with: what is the right level of detail of real-world factors we need to capture in our model? How do we prevent state spaces from exploding beyond practical computational tractability? The answers to these questions typically have to do with depth of understanding of the nuances of the real-world problem and a trial-and-error process of: formulating the model, solving for the optimal policy, testing the efficacy of this policy in practice (with appropriate measurements to capture real-world metrics), learning about the drawbacks of our model, and iterating back to tweak (or completely change) the model.

Let's consider a classic example of a card game such as Poker or Blackjack as a POMDP where your objective as a player is to identify the optimal policy to maximize your expected return (Optimal Value Function). The observation O_t would be the entire set of information you would have seen up to time step t (or a compressed version of this entire information that suffices for predicting transitions and for taking actions). The state S_t would include, among other things, the set of cards you have, the set of cards your opponents have (which you don't see), and the entire set of exposed as well as unexposed cards not held by players. Thus, the state is only partially observable. With this POMDP structure, we proceed to develop a model of the transition probabilities of next state S_{t+1} and reward R_{t+1}, conditional on current state S_t and current action A_t. We also develop a model of the probabilities of next observation O_{t+1}, conditional on next state S_{t+1} and current action A_t. These probabilities are estimated from data collected from various games (capturing opponent behaviors) and knowledge of the cards-structure of the deck (or decks) used to play the game. Now let's think about what would happen if we modeled this card game as an MDP. We'd no longer have the unseen cards as part of our state. Instead, the state S_t will be limited to the information seen upto time t (i.e., $S_t = O_t$). We can still estimate the transition probabilities, but since it's much harder to estimate in this case, our estimate will likely be quite noisy and nowhere near as reliable as the probability estimates in the POMDP case. The advantage though with modeling it as an MDP is that the algorithm to arrive at the Optimal Value Function/Optimal Policy is a lot more tractable compared to the algorithm for the POMDP model. So it's a tradeoff between the reliability of the probability estimates versus the tractability of the algorithm to solve for the Optimal Value Function/Policy.

The purpose of this subsection on POMDPs is to highlight that by default a lot of problems in the real-world are POMDPs and it can sometimes take quite a bit of domain-knowledge, modeling creativity and real-world experimentation to treat them as MDPs and make the solution to the modeled MDP successful in practice.

The idea of partial observability was introduced in a paper by K.J.Astrom (Åström 1965). To learn more about POMDP theory, we refer you to the POMDP book by Vikram Krishnamurthy (Krishnamurthy 2016).

1.12 Summary of Key Learnings from this Chapter

- MDP Bellman Policy Equations
- MDP Bellman Optimality Equations
- Theorem (1.10.1) on the existence of an Optimal Policy, and of each Optimal Policy achieving the Optimal Value Function

Bibliography

Åström, K. J. 1965. "Optimal Control of Markov Processes with Incomplete State Information." *Journal of Mathematical Analysis and Applications* 10 (1): 174–205. https://doi.org/10.1016/0022-247X(65)90154-X.

Bellman, Richard. 1957a. "A Markovian Decision Process." *Journal of Mathematics and Mechanics* 6 (5): 679–84. http://www.jstor.org/stable/24900506.

———. 1957b. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press.

Black, Fisher, and Myron S. Scholes. 1973. "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy* 81 (3): 637–54.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press.

Krishnamurthy, Vikram. 2016. *Partially Observed Markov Decision Processes: From Filtering to Controlled Sensing*. Cambridge University Press. https://doi.org/10.1017/CBO9781316471104.

Merton, Robert C. 1969. "Lifetime Portfolio Selection Under Uncertainty: The Continuous-Time Case." *The Review of Economics and Statistics* 51 (3): 247–57. https://doi.org/10.2307/1926560.

Puterman, Martin L. 2014. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.