Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Dynamic Programming Algorithms

As a reminder, much of this book is about algorithms to solve the MDP Control problem, i.e., to compute the Optimal Value Function (and an associated Optimal Policy). We will also cover algorithms for the MDP Prediction problem, i.e., to compute the Value Function when the AI agent executes a fixed policy π (which, as we know from Chapter **??**, is the same as computing the Value Function of the π-implied MRP). Our typical approach will be to first cover algorithms to solve the Prediction problem before covering algorithms to solve the Control problem - not just because Prediction is a key component in solving the Control problem, but also because it helps understand the key aspects of the techniques employed in the Control algorithm in the simpler setting of Prediction.

1.1 Planning versus Learning

In this book, we shall look at Prediction and Control from the lens of AI (and we'll specifically use the terminology of AI). We shall distinguish between algorithms that don't have a model of the MDP environment (no access to the $\mathcal{P}_R$ function) versus algorithms that do have a model of the MDP environment (meaning $\mathcal{P}_R$ is available to us either in terms of explicit probability distribution representations or available to us just as a sampling model). The former (algorithms without access to a model) are known as *Learning Algorithms* to reflect the fact that the AI agent will need to interact with the real-world environment (eg: a robot learning to navigate in an actual forest) and learn the Value Function from data (states encountered, actions taken, rewards observed) it receives through interactions with the environment. The latter (algorithms with access to a model of the MDP environment) are known as *Planning Algorithms* to reflect the fact that the AI agent requires no real-world environment interaction and in fact, projects (with the help of the model) probabilistic scenarios of future states/rewards for various choices of actions, and solves for the requisite Value Function based on the projected outcomes. In both Learning and Planning, the Bellman Equation is the fundamental concept driving the algorithms but the details of the algorithms will typically make them appear fairly different. We will only focus on Planning algorithms in this chapter, and in fact, will only focus on a subclass of Planning algorithms known as Dynamic Programming.

1.2 Usage of the term *Dynamic Programming*

Unfortunately, the term Dynamic Programming tends to be used by different fields in somewhat different ways. So it pays to clarify the history and the current usage of the term. The term *Dynamic Programming* was coined by Richard Bellman himself. Here is the rather interesting story told by Bellman about how and why he coined the term.

> "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for

> mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

Bellman had coined the term Dynamic Programming to refer to the general theory of MDPs, together with the techniques to solve MDPs (i.e., to solve the Control problem). So the MDP Bellman Optimality Equation was part of this catch-all term *Dynamic Programming*. The core semantic of the term Dynamic Programming was that the Optimal Value Function can be expressed recursively - meaning, to act optimally from a given state, we will need to act optimally from each of the resulting next states (which is the essence of the Bellman Optimality Equation). In fact, Bellman used the term "Principle of Optimality" to refer to this idea of "Optimal Substructure," and articulated it as follows:

> PRINCIPLE OF OPTIMALITY. An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.

So, you can see that the term Dynamic Programming was not just an algorithm in its original usage. Crucially, Bellman laid out an iterative algorithm to solve for the Optimal Value Function (i.e., to solve the MDP Control problem). Over the course of the next decade, the term Dynamic Programming got associated with (multiple) algorithms to solve the MDP Control problem. The term Dynamic Programming was extended to also refer to algorithms to solve the MDP Prediction problem. Over the next couple of decades, Computer Scientists started refering to the term Dynamic Programming as any algorithm that solves a problem through a recursive formulation as long as the algorithm makes repeated invocations to the solutions of each subproblem (overlapping subproblem structure). A classic such example is the algorithm to compute the Fibonacci sequence by caching the Fibonacci values and re-using those values during the course of the algorithm execution. The algorithm to calculate the shortest path in a graph is another classic example where each shortest (i.e. optimal) path includes sub-paths that are optimal. However, in this book, we won't use the term Dynamic Programming in this broader sense. We will

use the term Dynamic Programming to be restricted to algorithms to solve the MDP Prediction and Control problems (even though Bellman originally used it only in the context of Control). More specifically, we will use the term Dynamic Programming in the narrow context of Planning algorithms for problems with the following two specializations:

- The state space is finite, the action space is finite, and the set of pairs of next state and reward (given any pair of current state and action) are also finite.
- We have explicit knowledge of the model probabilities (either in the form of $\mathcal{P}_R$ or in the form of $\mathcal{P}$ and $\mathcal{R}$ separately).

This is the setting of the class `FiniteMarkovDecisionProcess` we had covered in Chapter **??**. In this setting, Dynamic Programming algorithms solve the Prediction and Control problems *exactly* (meaning the computed Value Function converges to the true Value Function as the algorithm iterations keep increasing). There are variants of Dynamic Programming algorithms known as Asynchronous Dynamic Programming algorithms, Approximate Dynamic Programming algorithms etc. But without such qualifications, when we use just the term Dynamic Programming, we will be refering to the "classical" iterative algorithms (that we will soon describe) for the above-mentioned setting of the `FiniteMarkovDecisionProcess` class to solve MDP Prediction and Control *exactly*. Even though these classical Dynamical Programming algorithms don't scale to large state/action spaces, they are extremely vital to develop one's core understanding of the key concepts in the more advanced algorithms that will enable us to scale (eg: the Reinforcement Learning algorithms that we shall introduce in later chapters).

1.3 Fixed-Point Theory

We start by covering 3 classical Dynamic Programming algorithms. Each of the 3 algorithms is founded on the Bellman Equations we had covered in Chapter **??**. Each of the 3 algorithms is an iterative algorithm where the computed Value Function converges to the true Value Function as the number of iterations approaches infinity. Each of the 3 algorithms is based on the concept of *Fixed-Point* and updates the computed Value Function towards the Fixed-Point (which in this case, is the true Value Function). Fixed-Point is actually a fairly generic and important concept in the broader fields of Pure as well as Applied Mathematics (also important in Theoretical Computer Science), and we believe understanding Fixed-Point theory has many benefits beyond the needs of the subject of this book. Of more relevance is the fact that the Fixed-Point view of Dynamic Programming is the best way to understand Dynamic Programming. We shall not only cover the theory of Dynamic Programming through the Fixed-Point perspective, but we shall also implement Dynamic Programming algorithms in our code based on the Fixed-Point concept. So this section will be a short primer on general Fixed-Point Theory (and implementation in code) before we get to the 3 Dynamic Programming algorithms.

Definition 1.3.1. The Fixed-Point of a function $f : \mathcal{X} \to \mathcal{X}$ (for some arbitrary domain $\mathcal{X}$) is a value $x \in \mathcal{X}$ that satisfies the equation: $x = f(x)$.

Note that for some functions, there will be multiple fixed-points and for some other functions, a fixed-point won't exist. We will be considering functions which have a unique fixed-point (this will be the case for the Dynamic Programming algorithms).

Let's warm up to the above-defined abstract concept of Fixed-Point with a concrete example. Consider the function $f(x) = \cos(x)$ defined for $x \in \mathbb{R}$ (x in radians, to be clear).

So we want to solve for an x such that $x = \cos(x)$. Knowing the frequency and amplitude of cosine, we can see that the cosine curve intersects the line $y = x$ at only one point, which should be somewhere between 0 and $\frac{\pi}{2}$. But there is no easy way to solve for this point. Here's an idea: Start with any value $x_0 \in \mathbb{R}$, calculate $x_1 = \cos(x_0)$, then calculate $x_2 = \cos(x_1)$, and so on ..., i.e, $x_{i+1} = \cos(x_i)$ for $i = 0, 1, 2, \ldots$. You will find that x_i and x_{i+1} get closer and closer as i increases, i.e., $|x_{i+1} - x_i| \leq |x_i - x_{i-1}|$ for all $i \geq 1$. So it seems like $\lim_{i\to\infty} x_i = \lim_{i\to\infty} \cos(x_{i-1}) = \lim_{i\to\infty} \cos(x_i)$, which would imply that for large enough i, x_i would serve as an approximation to the solution of the equation $x = \cos(x)$. But why does this method of repeated applications of the function f (no matter what x_0 we start with) work? Why does it not diverge or oscillate? How quickly does it converge? If there were multiple fixed-points, which fixed-point would it converge to (if at all)? Can we characterize a class of functions f for which this method (repeatedly applying f, starting with any arbitrary value of x_0) would work (in terms of solving the equation $x = f(x)$)? These are the questions Fixed-Point theory attempts to answer. Can you think of problems you have solved in the past which fall into this method pattern that we've illustrated above for $f(x) = \cos(x)$? It's likely you have, because most of the root-finding and optimization methods (including multi-variate solvers) are essentially based on the idea of Fixed-Point. If this doesn't sound convincing, consider the simple Newton method:

For a differentiable function $g : \mathbb{R} \to \mathbb{R}$ whose root we want to solve for, the Newton method update rule is:

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

Setting $f(x) = x - \frac{g(x)}{g'(x)}$, the update rule is:

$$x_{i+1} = f(x_i)$$

and it solves the equation $x = f(x)$ (solves for the fixed-point of f), i.e., it solves the equation:

$$x = x - \frac{g(x)}{g'(x)} \Rightarrow g(x) = 0$$

Thus, we see the same method pattern as we saw above for $\cos(x)$ (repeated application of a function, starting with any initial value) enables us to solve for the root of g.

More broadly, what we are saying is that if we have a function $f : \mathcal{X} \to \mathcal{X}$ (for some arbitrary domain $\mathcal{X}$), under appropriate conditions (that we will state soon), $f(f(\ldots f(x_0)\ldots))$ converges to a fixed-point of f, i.e., to the solution of the equation $x = f(x)$ (no matter what $x_0 \in \mathcal{X}$ we start with). Now we are ready to state this formally. The statement of the following theorem (due to Stefan Banach) is quite terse, so we will provide plenty of explanation on how to interpret it and how to use it after stating the theorem (we skip the proof of the theorem).

Theorem 1.3.1 (Banach Fixed-Point Theorem)**.** *Let $\mathcal{X}$ be a non-empty set equipped with a complete metric $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Let $f : \mathcal{X} \to \mathcal{X}$ be such that there exists a $L \in [0, 1)$ such that $d(f(x_1), f(x_2)) \leq L \cdot d(x_1, x_2)$ for all $x_1, x_2 \in \mathcal{X}$ (this property of f is called a contraction, and we refer to f as a contraction function). Then,*

1. *There exists a unique Fixed-Point $x^* \in \mathcal{X}$, i.e.,*

$$x^* = f(x^*)$$

2. *For any* $x_0 \in \mathcal{X}$*, and sequence* $[x_i | i = 0, 1, 2, \ldots]$ *defined as* $x_{i+1} = f(x_i)$ *for all* $i = 0, 1, 2, \ldots$*,*

$$\lim_{i \to \infty} x_i = x^*$$

3.

$$d(x^*, x_i) \leq \frac{L^i}{1 - L} \cdot d(x_1, x_0)$$

Equivalently,

$$d(x^*, x_{i+1}) \leq \frac{L}{1 - L} \cdot d(x_{i+1}, x_i)$$
$$d(x^*, x_{i+1}) \leq L \cdot d(x^*, x_i)$$

We realize this is quite terse and will now demystify the theorem in a simple, intuitive manner. First we need to explain what *complete metric* means. Let's start with the term *metric*. A metric is simply a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that satisfies the usual "distance" properties (for any $x_1, x_2, x_3 \in \mathcal{X}$):

1. $d(x_1, x_2) = 0 \Leftrightarrow x_1 = x_2$ (meaning two different points have a distance strictly greater than 0)
2. $d(x_1, x_2) = d(x_2, x_1)$ (meaning distance is directionless)
3. $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$ (meaning the triangle inequality is satisfied)

The term *complete* is a bit of a technical detail on sequences not escaping the set $\mathcal{X}$ (that's required in the proof). Since we won't be doing the proof and since this technical detail is not so important for the intuition, we skip the formal definition of *complete*. A non-empty set $\mathcal{X}$ equipped with the function d (and the technical detail of being *complete*) is known as a complete metric space.

Now we move on to the key concept of *contraction*. A function $f : \mathcal{X} \to \mathcal{X}$ is said to be a contraction function if two points in $\mathcal{X}$ get closer when they are mapped by f (the statement: $d(f(x_1), f(x_2)) \leq L \cdot d(x_1, x_2)$ for all $x_1, x_2 \in \mathcal{X}$, for some $L \in [0, 1)$).

The theorem basically says that for any contraction function f, there is not only a unique fixed-point x^*, one can arrive at x^* by repeated application of f, starting with any initial value $x_0 \in \mathcal{X}$:

$$f(f(\ldots f(x_0) \ldots)) \to x^*$$

We use the notation $f^i : \mathcal{X} \to \mathcal{X}$ for $i = 0, 1, 2, \ldots$ as follows:

$$f^{i+1}(x) = f(f^i(x)) \text{ for all } i = 0, 1, 2, \ldots, \text{ for all } x \in \mathcal{X}$$
$$f^0(x) = x \text{ for all } x \in \mathcal{X}$$

With this notation, the computation of the fixed-point can be expressed as:

$$\lim_{i \to \infty} f^i(x_0) = x^* \text{ for all } x_0 \in \mathcal{X}$$

The algorithm, in iterative form, is:

$$x_{i+1} = f(x_i) \text{ for all } i = 0, 2, \ldots$$

We stop the algorithm when x_i and x_{i+1} are close enough based on the distance-metric d.

Banach Fixed-Point Theorem also gives us a statement on the speed of convergence relating the distance between x^* and any x_i to the distance between any two successive x_i.

This is a powerful theorem. All we need to do is identify the appropriate set $\mathcal{X}$ to work with, identify the appropriate metric d to work with, and ensure that f is indeed a contraction function (with respect to d). This enables us to solve for the fixed-point of f with the above-described iterative process of applying f repeatedly, starting with any arbitrary value of $x_0 \in \mathcal{X}$.

We leave it to you as an exercise to verify that $f(x) = \cos(x)$ is a contraction function in the domain $\mathcal{X} = \mathbb{R}$ with metric d defined as $d(x_1, x_2) = |x_1 - x_2|$. Now let's write some code to implement the fixed-point algorithm we described above. Note that we implement this for any generic type `X` to represent an arbitrary domain $\mathcal{X}$.

```
X = TypeVar('X')

def iterate(step: Callable[[X], X], start: X) -> Iterator[X]:
    state = start

    while True:
        yield state
        state = step(state)
```

The above function takes as input a function (`step: Callable[[X], X]`) and a starting value (`start: X`), and repeatedly applies the function while `yield`ing the values in the form of an `Iterator[X]`, i.e., as a stream of values. This produces an endless stream though. We need a way to specify convergence, i.e., when successive values of the stream are "close enough."

```
def converge(values: Iterator[X], done: Callable[[X, X], bool]) -> Iterator[X]:
    a = next(values, None)
    if a is None:
        return

    yield a

    for b in values:
        yield b
        if done(a, b):
            return

        a = b
```

The above function `converge` takes as input the generated values from `iterate` (argument `values: Iterator[X]`) and a signal to indicate convergence (argument `done: Callable[[X, X], bool]`), and produces the generated values until `done` is `True`. It is the user's responsibility to write the function `done` and pass it to `converge`. Now let's use these two functions to solve for $x = \cos(x)$.

```
import numpy as np
x = 0.0
values = converge(
    iterate(lambda y: np.cos(y), x),
    lambda a, b: np.abs(a - b) < 1e-3
)
for i, v in enumerate(values):
    print(f"{i}: {v:.4f}")
```

This prints a trace with the index of the stream and the value at that index as the function cos is repeatedly applied. It terminates when two successive values are within 3 decimal places of each other.

```
0: 0.0000
1: 1.0000
2: 0.5403
3: 0.8576
4: 0.6543
5: 0.7935
6: 0.7014
7: 0.7640
8: 0.7221
9: 0.7504
10: 0.7314
11: 0.7442
12: 0.7356
13: 0.7414
14: 0.7375
15: 0.7401
16: 0.7384
17: 0.7396
18: 0.7388
```

We encourage you to try other starting values (other than the one we have above: $x_0 = 0.0$) and see the trace. We also encourage you to identify other functions f which are contractions in an appropriate metric. The above fixed-point code is in the file rl/iterate.py. In this file, you will find two more functions `last` and `converged` to produce the final value of the given iterator when it's values converge according to the `done` function.

1.4 Bellman Policy Operator and Policy Evaluation Algorithm

Our first Dynamic Programming algorithm is called *Policy Evaluation*. The Policy Evaluation algorithm solves the problem of calculating the Value Function of a Finite MDP evaluated with a fixed policy π (i.e., the Prediction problem for finite MDPs). We know that this is equivalent to calculating the Value Function of the π-implied Finite MRP. To avoid notation confusion, note that a superscript of π for a symbol means it refers to notation for the π-implied MRP. The precise specification of the Prediction problem is as follows:

Let the states of the MDP (and hence, of the π-implied MRP) be $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$, and without loss of generality, let $\mathcal{N} = \{s_1, s_2, \ldots, s_m\}$ be the non-terminal states. We are given a fixed policy $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$. We are also given the π-implied MRP's transition probability function:

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

in the form of a data structure (since the states are finite, and the pairs of next state and reward transitions from each non-terminal state are also finite). The Prediction problem is to compute the Value Function of the MDP when evaluated with the policy π (equivalently, the Value Function of the π-implied MRP), which we denote as $V^\pi : \mathcal{N} \to \mathbb{R}$.

We know from Chapters **??** and **??** that by extracting (from $\mathcal{P}_R^\pi$) the transition probability function $\mathcal{P}^\pi : \mathcal{N} \times \mathcal{S} \to [0, 1]$ of the implicit Markov Process and the reward function

$\mathcal{R}^\pi : \mathcal{N} \to \mathbb{R}$, we can perform the following calculation for the Value Function $V^\pi : \mathcal{N} \to \mathbb{R}$ (expressed as a column vector $\boldsymbol{V}^\pi \in \mathbb{R}^m$) to solve this Prediction problem:

$$\boldsymbol{V}^\pi = (\boldsymbol{I_m} - \gamma \boldsymbol{\mathcal{P}}^\pi)^{-1} \cdot \boldsymbol{\mathcal{R}}^\pi$$

where $\boldsymbol{I_m}$ is the $m \times m$ identity matrix, column vector $\boldsymbol{\mathcal{R}}^\pi \in \mathbb{R}^m$ represents $\mathcal{R}^\pi$, and $\boldsymbol{\mathcal{P}}^\pi$ is an $m \times m$ matrix representing $\mathcal{P}^\pi$ (rows and columns corresponding to the non-terminal states). However, when m is large, this calculation won't scale. So, we look for a numerical algorithm that would solve (for $\boldsymbol{V}^\pi$) the following MRP Bellman Equation (for a larger number of finite states).

$$\boldsymbol{V}^\pi = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V}^\pi$$

We define the *Bellman Policy Operator* $\boldsymbol{B}^\pi : \mathbb{R}^m \to \mathbb{R}^m$ as:

$$\boldsymbol{B}^\pi(\boldsymbol{V}) = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V} \text{ for any vector } \boldsymbol{V} \text{ in the vector space } \mathbb{R}^m \tag{1.1}$$

So, the MRP Bellman Equation can be expressed as:

$$\boldsymbol{V}^\pi = \boldsymbol{B}^\pi(\boldsymbol{V}^\pi)$$

which means $\boldsymbol{V}^\pi \in \mathbb{R}^m$ is a Fixed-Point of the *Bellman Policy Operator* $\boldsymbol{B}^\pi : \mathbb{R}^m \to \mathbb{R}^m$. Note that the Bellman Policy Operator can be generalized to the case of non-finite MDPs and V^π is still a Fixed-Point for various generalizations of interest. However, since this chapter focuses on developing algorithms for finite MDPs, we will work with the above narrower (Equation (1.1)) definition. Also, for proofs of correctness of the DP algorithms (based on Fixed-Point) in this chapter, we shall assume the discount factor $\gamma < 1$.

Note that $\boldsymbol{B}^\pi$ is an affine transformation on vectors in $\mathbb{R}^m$ and should be thought of as a generalization of a simple 1-D ($\mathbb{R} \to \mathbb{R}$) affine transformation $y = a + bx$ where the multiplier b is replaced with the matrix $\gamma \boldsymbol{\mathcal{P}}^\pi$ and the shift a is replaced with the column vector $\boldsymbol{\mathcal{R}}^\pi$.

We'd like to come up with a metric for which $\boldsymbol{B}^\pi$ is a contraction function so we can take advantage of Banach Fixed-Point Theorem and solve this Prediction problem by iterative applications of the Bellman Policy Operator $\boldsymbol{B}^\pi$. For any Value Function $\boldsymbol{V} \in \mathbb{R}^m$ (representing $V : \mathcal{N} \to \mathbb{R}$), we shall express the Value for any state $s \in \mathcal{N}$ as $\boldsymbol{V}(s)$.

Our metric $d : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ shall be the L^∞ norm defined as:

$$d(\boldsymbol{X}, \boldsymbol{Y}) = \|\boldsymbol{X} - \boldsymbol{Y}\|_\infty = \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

$\boldsymbol{B}^\pi$ is a contraction function under L^∞ norm because for all $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$,

$$\max_{s \in \mathcal{N}} |(\boldsymbol{B}^\pi(\boldsymbol{X}) - \boldsymbol{B}^\pi(\boldsymbol{Y}))(s)| = \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{\mathcal{P}}^\pi \cdot (\boldsymbol{X} - \boldsymbol{Y}))(s)| \leq \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

So invoking Banach Fixed-Point Theorem proves the following Theorem:

Theorem 1.4.1 (Policy Evaluation Convergence Theorem)**.** *For a Finite MDP with $|\mathcal{N}| = m$ and $\gamma < 1$, if $\boldsymbol{V}^\pi \in \mathbb{R}^m$ is the Value Function of the MDP when evaluated with a fixed policy $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$, then $\boldsymbol{V}^\pi$ is the unique Fixed-Point of the Bellman Policy Operator $\boldsymbol{B}^\pi : \mathbb{R}^m \to \mathbb{R}^m$, and*

$$\lim_{i \to \infty} (\boldsymbol{B}^\pi)^i(\boldsymbol{V_0}) \to \boldsymbol{V}^\pi \text{ for all starting Value Functions } \boldsymbol{V_0} \in \mathbb{R}^m$$

This gives us the following iterative algorithm (known as the *Policy Evaluation* algorithm for fixed policy $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$):

- Start with any Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$
- Iterating over $i = 0, 1, 2, \ldots$, calculate in each iteration:

$$\boldsymbol{V_{i+1}} = \boldsymbol{B}^\pi(\boldsymbol{V_i}) = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V_i}$$

We stop the algorithm when $d(\boldsymbol{V_i}, \boldsymbol{V_{i+1}}) = \max_{s \in \mathcal{N}} |(\boldsymbol{V_i} - \boldsymbol{V_{i+1}})(s)|$ is adequately small.

It pays to emphasize that Banach Fixed-Point Theorem not only assures convergence to the unique solution $\boldsymbol{V}^\pi$ (no matter what Value Function $\boldsymbol{V_0}$ we start the algorithm with), it also assures a reasonable speed of convergence (dependent on the choice of starting Value Function $\boldsymbol{V_0}$ and the choice of γ). Now let's write the code for Policy Evaluation.

```
DEFAULT_TOLERANCE = 1e-5
V = Mapping[NonTerminal[S], float]

def evaluate_mrp(
    mrp: FiniteMarkovRewardProcess[S],
    gamma: float
) -> Iterator[np.ndarray]:
    def update(v: np.ndarray) -> np.ndarray:
        return mrp.reward_function_vec + gamma * \
            mrp.get_transition_matrix().dot(v)

    v_0: np.ndarray = np.zeros(len(mrp.non_terminal_states))

    return iterate(update, v_0)

def almost_equal_np_arrays(
    v1: np.ndarray,
    v2: np.ndarray,
    tolerance: float = DEFAULT_TOLERANCE
) -> bool:
    return max(abs(v1 - v2)) < tolerance

def evaluate_mrp_result(
    mrp: FiniteMarkovRewardProcess[S],
    gamma: float
) -> V[S]:
    v_star: np.ndarray = converged(
        evaluate_mrp(mrp, gamma=gamma),
        done=almost_equal_np_arrays
    )
    return {s: v_star[i] for i, s in enumerate(mrp.non_terminal_states)}
```

The code should be fairly self-explanatory. Since the Policy Evaluation problem applies to Finite MRPs, the function `evaluate_mrp` above takes as input `mrp: FiniteMarkovDecisionProcess[S]` and a `gamma: float` to produce an `Iterator` on Value Functions represented as `np.ndarray` (for fast vector/matrix calculations). The function `update` in `evaluate_mrp` represents the application of the Bellman Policy Operator $\boldsymbol{B}^\pi$. The function `evaluate_mrp_result` produces the Value Function for the given `mrp` and the given `gamma`, returning the last value function on the `Iterator` (which terminates based on the `almost_equal_np_arrays` function, considering the maximum of the absolute value differences across all states). Note that the return type of `evaluate_mrp_result` is `V[S]` which is an alias for `Mapping[NonTerminal[S], float]`, capturing the semantic of $\mathcal{N} \to \mathbb{R}$. Note that `evaluate_mrp` is useful for debugging (by looking at the trace of value functions in the execution of the Policy Evaluation algorithm) while `evaluate_mrp_result` produces the desired output Value Function.

Note that although we defined the Bellman Policy Operator $\boldsymbol{B}^\pi$ as operating on Value Functions of the π-implied MRP, we can also view the Bellman Policy Operator $\boldsymbol{B}^\pi$ as

operating on Value Functions of an MDP. To support this MDP view, we express Equation (1.1) in terms of the MDP transitions/rewards specification, as follows:

$$\boldsymbol{B}^{\pi}(\boldsymbol{V})(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \cdot \mathcal{R}(s,a) + \gamma \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V}(s') \text{ for all } s \in \mathcal{N} \quad (1.2)$$

If the number of non-terminal states of a given MRP is m, then the running time of each iteration is $O(m^2)$. Note though that to construct an MRP from a given MDP and a given policy, we have to perform $O(m^2 \cdot k)$ operations, where $k = |\mathcal{A}|$.

1.5 Greedy Policy

We had said earlier that we will be presenting 3 Dynamic Programming Algorithms. The first (Policy Evaluation), as we saw in the previous section, solves the MDP Prediction problem. The other two (that will present in the next two sections) solve the MDP Control problem. This section is a stepping stone from *Prediction* to *Control*. In this section, we define a function that is motivated by the idea of *improving a value function/improving a policy* with a "greedy" technique. Formally, the *Greedy Policy Function*

$$G : \mathbb{R}^m \to (\mathcal{N} \to \mathcal{A})$$

interpreted as a function mapping a Value Function $\boldsymbol{V}$ (represented as a vector) to a deterministic policy $\pi'_D : \mathcal{N} \to \mathcal{A}$, is defined as:

$$G(\boldsymbol{V})(s) = \pi'_D(s) = \arg\max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V}(s')\} \text{ for all } s \in \mathcal{N} \quad (1.3)$$

Note that for any specific s, if two or more actions a achieve the maximization of $\mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V}(s')$, then we use an arbitrary rule in breaking ties and assigning a single action a as the output of the above arg max operation. We shall use Equation (1.3) in our mathematical exposition but we require a different (but equivalent) expression for $G(\boldsymbol{V})(s)$ to guide us with our code since the interface for `FiniteMarkovDecisionProcess` operates on $\mathcal{P}_R$, rather than $\mathcal{R}$ and $\mathcal{P}$. The equivalent expression for $G(\boldsymbol{V})(s)$ is as follows:

$$G(\boldsymbol{V})(s) = \arg\max_{a \in \mathcal{A}} \{\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{D}} \mathcal{P}_R(s,a,r,s') \cdot (r + \gamma \cdot \boldsymbol{W}(s'))\} \text{ for all } s \in \mathcal{N} \quad (1.4)$$

where $\boldsymbol{W} \in \mathbb{R}^n$ is defined as:

$$\boldsymbol{W}(s') = \begin{cases} \boldsymbol{V}(s') & \text{if } s' \in \mathcal{N} \\ 0 & \text{if } s' \in \mathcal{T} = \mathcal{S} - \mathcal{N} \end{cases}$$

Note that in Equation (1.4), because we have to work with $\mathcal{P}_R$, we need to consider transitions to all states $s' \in \mathcal{S}$ (versus transition to all states $s' \in \mathcal{N}$ in Equation (1.3)), and so, we need to handle the transitions to states $s' \in \mathcal{T}$ carefully (essentially by using the $\boldsymbol{W}$ function as described above).

Now let's write some code to create this "greedy policy" from a given value function, guided by Equation (1.4).

```
import operator

def extended_vf(v: V[S], s: State[S]) -> float:
    def non_terminal_vf(st: NonTerminal[S], v=v) -> float:
        return v[st]
    return s.on_non_terminal(non_terminal_vf, 0.0)

def greedy_policy_from_vf(
    mdp: FiniteMarkovDecisionProcess[S, A],
    vf: V[S],
    gamma: float
) -> FiniteDeterministicPolicy[S, A]:
    greedy_policy_dict: Dict[S, A] = {}

    for s in mdp.non_terminal_states:
        q_values: Iterator[Tuple[A, float]] = \
            ((a, mdp.mapping[s][a].expectation(
                lambda s_r: s_r[1] + gamma * extended_vf(vf, s_r[0])
            )) for a in mdp.actions(s))
        greedy_policy_dict[s.state] = \
            max(q_values, key=operator.itemgetter(1))[0]

    return FiniteDeterministicPolicy(greedy_policy_dict)
```

As you can see above, the function `greedy_policy_from_vf` loops through all the non-terminal states that serve as keys in `greedy_policy_dict: Dict[S, A]`. Within this loop, we go through all the actions in $\mathcal{A}(s)$ and compute Q-Value $Q(s,a)$ as the sum (over all (s',r) pairs) of $\mathcal{P}_R(s,a,r,s') \cdot (r + \gamma \cdot \boldsymbol{W}(s'))$, written as $\mathbb{E}_{(s',r)\sim\mathcal{P}_R}[r + \gamma \cdot \boldsymbol{W}(s')]$. Finally, we calculate $\arg\max_a Q(s,a)$ for all non-terminal states s, and return it as a `FinitePolicy` (which is our greedy policy).

Note that the `extended_vf` represents the $\boldsymbol{W} : \mathcal{S} \to \mathbb{R}$ function used in the right-hand-side of Equation (1.4), which is the usual value function when it's argument is a non-terminal state and is the default value of 0 when it's argument is a terminal state. We shall use the `extended_vf` function in other Dynamic Programming algorithms later in this chapter as they also involve the $\boldsymbol{W} : \mathcal{S} \to \mathbb{R}$ function in the right-hand-side of their corresponding governing equation.

The word "Greedy" is a reference to the term "Greedy Algorithm," which means an algorithm that takes heuristic steps guided by locally-optimal choices in the hope of moving towards a global optimum. Here, the reference to *Greedy Policy* means if we have a policy π and its corresponding Value Function $\boldsymbol{V}^\pi$ (obtained say using Policy Evaluation algorithm), then applying the Greedy Policy function G on $\boldsymbol{V}^\pi$ gives us a deterministic policy $\pi'_D : \mathcal{N} \to \mathcal{A}$ that is hopefully "better" than π in the sense that $\boldsymbol{V}^{\pi'_D}$ is "greater" than $\boldsymbol{V}^\pi$. We shall now make this statement precise and show how to use the *Greedy Policy Function* to perform *Policy Improvement*.

1.6 Policy Improvement

Terms such as "better" or "improvement" refer to either Value Functions or to Policies (in the latter case, to Value Functions of an MDP evaluated with the policies). So what does it mean to say a Value Function $X : \mathcal{N} \to \mathbb{R}$ is "better" than a Value Function $Y : \mathcal{N} \to \mathbb{R}$? Here's the answer:

Definition 1.6.1 (Value Function Comparison)**.** We say $X \geq Y$ for Value Functions $X, Y : \mathcal{N} \to \mathbb{R}$ of an MDP if and only if:

$$X(s) \geq Y(s) \text{ for all } s \in \mathcal{N}$$

If we are dealing with finite MDPs (with m non-terminal states), we'd represent the Value Functions as vector $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$, and say that $\boldsymbol{X} \geq \boldsymbol{Y}$ if and only if $\boldsymbol{X}(s) \geq \boldsymbol{Y}(s)$ for all $s \in \mathcal{N}$.

So whenever you hear terms like "Better Value Function" or "Improved Value Function," you should interpret it to mean that the Value Function is *no worse for each of the states* (versus the Value Function it's being compared to).

So then, what about the claim of $\pi'_D = G(\boldsymbol{V}^\pi)$ being "better" than π? The following important theorem by Richard Bellman (Bellman 1957b) provides the clarification:

Theorem 1.6.1 (Policy Improvement Theorem)**.** *For a finite MDP, for any policy π,*

$$\boldsymbol{V}^{\pi'_D} = \boldsymbol{V}^{G(\boldsymbol{V}^\pi)} \geq \boldsymbol{V}^\pi$$

Proof. This proof is based on application of the Bellman Policy Operator on Value Functions of the given MDP (note: this MDP view of the Bellman Policy Operator is expressed in Equation (1.2)). We start by noting that applying the Bellman Policy Operator $\boldsymbol{B}^{\pi'_D}$ repeatedly, starting with the Value Function $\boldsymbol{V}^\pi$, will converge to the Value Function $\boldsymbol{V}^{\pi'_D}$. Formally,

$$\lim_{i \to \infty} (\boldsymbol{B}^{\pi'_D})^i (\boldsymbol{V}^\pi) = \boldsymbol{V}^{\pi'_D}$$

So the proof is complete if we prove that:

$$(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^\pi) \geq (\boldsymbol{B}^{\pi'_D})^i (\boldsymbol{V}^\pi) \text{ for all } i = 0, 1, 2, \ldots$$

which means we get a non-decreasing sequence of Value Functions $[(\boldsymbol{B}^{\pi'_D})^i(\boldsymbol{V}^\pi) | i = 0, 1, 2, \ldots]$ with repeated applications of $\boldsymbol{B}^{\pi'_D}$ starting with the Value Function $\boldsymbol{V}^\pi$.

Let us prove this by induction. The base case (for $i = 0$) of the induction is to prove that:

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi) \geq \boldsymbol{V}^\pi$$

Note that for the case of the deterministic policy π'_D and Value Function $\boldsymbol{V}^\pi$, Equation (1.2) simplifies to:

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi)(s) = \mathcal{R}(s, \pi'_D(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot \boldsymbol{V}^\pi(s') \text{ for all } s \in \mathcal{N}$$

From Equation (1.3), we know that for each $s \in \mathcal{N}$, $\pi'_D(s) = G(\boldsymbol{V}^\pi)(s)$ is the action that maximizes $\{\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V}^\pi(s')\}$. Therefore,

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

Let's compare this equation against the Bellman Policy Equation for π (below):

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

We see that $\boldsymbol{V}^\pi(s)$ is a weighted average of $Q^\pi(s, a)$ (with weights equal to probabilities $\pi(s, a)$ over choices of a) while $\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi)(s)$ is the maximum (over choices of a) of $Q^\pi(s, a)$. Therefore,

$$\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^\pi) \geq \boldsymbol{V}^\pi$$

This establishes the base case of the proof by induction. Now to complete the proof, all we have to do is to prove:

If $(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^{\pi})$, then $(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi})$ for all $i = 0, 1, 2, \ldots$

Since $(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi}) = \boldsymbol{B}^{\pi'_D}((\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^{\pi}))$, from the definition of Bellman Policy Operator (Equation (1.1)), we can write the following two equations:

$$(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^{\pi})(s) = \mathcal{R}(s, \pi'_D(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi})(s') \text{ for all } s \in \mathcal{N}$$

$$(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi})(s) = \mathcal{R}(s, \pi'_D(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^{\pi})(s') \text{ for all } s \in \mathcal{N}$$

Subtracting each side of the second equation from the first equation yields:

$$(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^{\pi})(s) - (\boldsymbol{B}^{\pi'_D})^{i+1}(s)$$
$$= \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, \pi'_D(s), s') \cdot ((\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi})(s') - (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^{\pi})(s'))$$

for all $s \in \mathcal{N}$

Since $\gamma \mathcal{P}(s, \pi'_D(s), s')$ consists of all non-negative values and since the induction step assumes $(\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi})(s') \geq (\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^{\pi})(s')$ for all $s' \in \mathcal{N}$, the right-hand-side of this equation is non-negative, meaning the left-hand-side of this equation is non-negative, i.e.,

$$(\boldsymbol{B}^{\pi'_D})^{i+2}(\boldsymbol{V}^{\pi})(s) \geq (\boldsymbol{B}^{\pi'_D})^{i+1}(\boldsymbol{V}^{\pi})(s) \text{ for all } s \in \mathcal{N}$$

This completes the proof by induction. □

The way to understand the above proof is to think in terms of how each stage of further application of $\boldsymbol{B}^{\pi'_D}$ improves the Value Function. Stage 0 is when you have the Value Function $\boldsymbol{V}^{\pi}$ where we execute the policy π throughout the MDP. Stage 1 is when you have the Value Function $\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi})$ where from each state s, we execute the policy π'_D for the first time step following s and then execute the policy π for all further time steps. This has the effect of improving the Value Function from Stage 0 ($\boldsymbol{V}^{\pi}$) to Stage 1 ($\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi})$). Stage 2 is when you have the Value Function $(\boldsymbol{B}^{\pi'_D})^2(\boldsymbol{V}^{\pi})$ where from each state s, we execute the policy π'_D for the first two time steps following s and then execute the policy π for all further time steps. This has the effect of improving the Value Function from Stage 1 ($\boldsymbol{B}^{\pi'_D}(\boldsymbol{V}^{\pi})$) to Stage 2 ($(\boldsymbol{B}^{\pi'_D})^2(\boldsymbol{V}^{\pi})$). And so on ... each stage applies policy π'_D instead of policy π for one extra time step, which has the effect of improving the Value Function. Note that "improve" means $\geq$ (really means that the Value Function doesn't get worse for *any* of the states). These stages are simply the iterations of the Policy Evaluation algorithm (using policy π'_D) with starting Value Function $\boldsymbol{V}^{\pi}$, building a non-decreasing sequence of Value Functions $[(\boldsymbol{B}^{\pi'_D})^{i}(\boldsymbol{V}^{\pi}) | i = 0, 1, 2, \ldots]$ that get closer and closer until they converge to the Value Function $\boldsymbol{V}^{\pi'_D}$ that is $\geq \boldsymbol{V}^{\pi}$ (hence, the term *Policy Improvement*).

The Policy Improvement Theorem yields our first Dynamic Programming algorithm (called *Policy Iteration*) to solve the MDP Control problem. The Policy Iteration algorithm is due to Ronald Howard (Howard 1960).

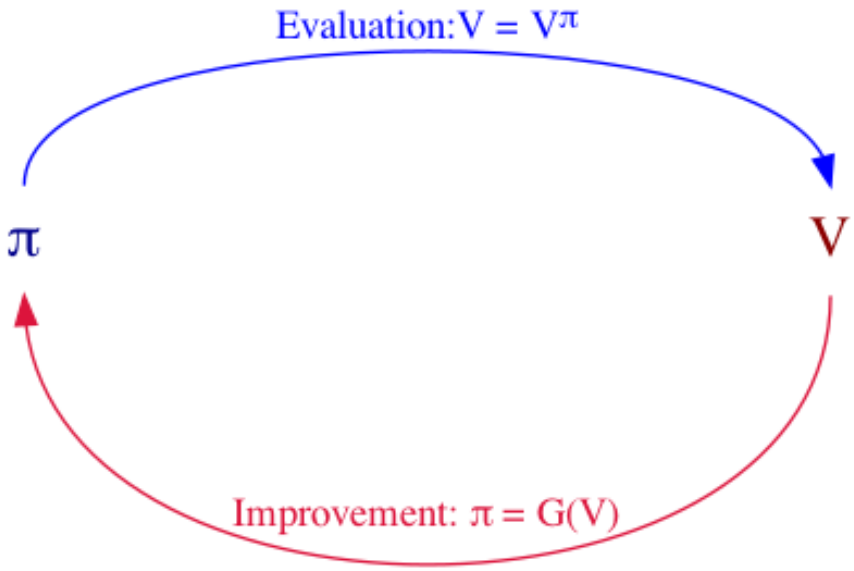

Figure 1.1: Policy Iteration Loop

1.7 Policy Iteration Algorithm

The proof of the Policy Improvement Theorem has shown us how to start with the Value Function $\boldsymbol{V}^{\pi}$ (for a policy π), perform a greedy policy improvement to create a policy $\pi'_D = G(\boldsymbol{V}^{\pi})$, and then perform Policy Evaluation (with policy π'_D) with starting Value Function $\boldsymbol{V}^{\pi}$, resulting in the Value Function $\boldsymbol{V}^{\pi'_D}$ that is an improvement over the Value Function $\boldsymbol{V}^{\pi}$ we started with. Now note that we can do the same process again to go from π'_D and $\boldsymbol{V}^{\pi'_D}$ to an improved policy π''_D and associated improved Value Function $\boldsymbol{V}^{\pi''_D}$. And we can keep going in this way to create further improved policies and associated Value Functions, until there is no further improvement. This methodology of performing Policy Improvement together with Policy Evaluation using the improved policy, in an iterative manner (depicted in Figure 1.1), is known as the Policy Iteration algorithm (shown below).

- Start with any Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$
- Iterating over $j = 0, 1, 2, \ldots$, calculate in each iteration:

$$\text{Deterministic Policy } \pi_{j+1} = G(\boldsymbol{V_j})$$

$$\text{Value Function } \boldsymbol{V_{j+1}} = \lim_{i \to \infty} (\boldsymbol{B}^{\pi_{j+1}})^i(\boldsymbol{V_j})$$

We perform these iterations (over j) until $\boldsymbol{V_{j+1}}$ is identical to $\boldsymbol{V_j}$ (i.e., there is no further improvement to the Value Function). When this happens, the following should hold:

$$\boldsymbol{V_j} = (\boldsymbol{B}^{G(\boldsymbol{V_j})})^i(\boldsymbol{V_j}) = \boldsymbol{V_{j+1}} \text{ for all } i = 0, 1, 2, \ldots$$

In particular, this equation should hold for $i = 1$:

$$\boldsymbol{V_j}(s) = \boldsymbol{B}^{G(\boldsymbol{V_j})}(\boldsymbol{V_j})(s) = \mathcal{R}(s, G(\boldsymbol{V_j})(s)) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s, G(\boldsymbol{V_j})(s), s') \cdot \boldsymbol{V_j}(s') \text{ for all } s \in \mathcal{N}$$

Figure 1.2: Policy Iteration Convergence

From Equation (1.3), we know that for each $s \in \mathcal{N}$, $\pi_{j+1}(s) = G(\boldsymbol{V_j})(s)$ is the action that maximizes $\{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V_j}(s')\}$. Therefore,

$$\boldsymbol{V_j}(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V_j}(s')\} \text{ for all } s \in \mathcal{N}$$

But this in fact is the MDP State-Value Function Bellman Optimality Equation, which would mean that $\boldsymbol{V_j} = \boldsymbol{V}^*$, i.e., when V_{j+1} is identical to V_j, the Policy Iteration algorithm has converged to the Optimal Value Function. The associated deterministic policy at the convergence of the Policy Iteration algorithm ($\pi_j : \mathcal{N} \to \mathcal{A}$) is an Optimal Policy because $\boldsymbol{V}^{\pi_j} = \boldsymbol{V_j} \approxeq \boldsymbol{V}^*$, meaning that evaluating the MDP with the deterministic policy π_j achieves the Optimal Value Function (depicted in Figure 1.2). This means the Policy Iteration algorithm solves the MDP Control problem. This proves the following Theorem:

Theorem 1.7.1 (Policy Iteration Convergence Theorem)**.** *For a Finite MDP with $|\mathcal{N}| = m$ and $\gamma < 1$, Policy Iteration algorithm converges to the Optimal Value Function $\boldsymbol{V}^* \in \mathbb{R}^m$ along with a Deterministic Optimal Policy $\pi^*_D : \mathcal{N} \to \mathcal{A}$, no matter which Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$ we start the algorithm with.*

Now let's write some code for Policy Iteration Algorithm. Unlike Policy Evaluation which repeatedly operates on Value Functions (and returns a Value Function), Policy Iteration repeatedly operates on a pair of Value Function and Policy (and returns a pair of Value Function and Policy). In the code below, notice the type `Tuple[V[S], FinitePolicy[S, A]]` that represents a pair of Value Function and Policy. The function `policy_iteration` repeatedly applies the function `update` on a pair of Value Function and Policy. The `update` function, after splitting its input `vf_policy` into `vf: V[S]` and `pi: FinitePolicy[S, A]`, creates an MRP (`mrp: FiniteMarkovRewardProcess[S]`) from the combination of the input `mdp` and `pi`. Then it performs a policy evaluation on `mrp` (using the `evaluate_mrp_result` function) to produce a Value Function `policy_vf: V[S]`, and finally creates a greedy (improved) policy named `improved_pi` from `policy_vf` (using the previously-written function `greedy_policy_from_vf`). Thus the function `update` performs a Policy Evaluation followed by a Policy Improvement. Notice also that `policy_iteration` offers the option to perform the linear-algebra-solver-based computation of Value Function for a given policy (`get_value_function_vec` method of the `mrp` object), in case the state space is not too large. `policy_iteration` returns an `Iterator` on pairs of Value Function and Policy produced by this process of repeated Policy Evaluation and Policy Improvement. `almost_equal_vf_pis` is the function to decide termination based on the distance between two successive Value Functions produced by Policy Iteration. `policy_iteration_result` returns the final (optimal) pair of Value Function and Policy (from the `Iterator` produced by `policy_iteration`), based on the termination criterion of `almost_equal_vf_pis`.

```
DEFAULT_TOLERANCE = 1e-5
```

```
def policy_iteration(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float,
    matrix_method_for_mrp_eval: bool = False
) -> Iterator[Tuple[V[S], FinitePolicy[S, A]]]:

    def update(vf_policy: Tuple[V[S], FinitePolicy[S, A]])\
            -> Tuple[V[S], FiniteDeterministicPolicy[S, A]]:

        vf, pi = vf_policy
        mrp: FiniteMarkovRewardProcess[S] = mdp.apply_finite_policy(pi)
        policy_vf: V[S] = {mrp.non_terminal_states[i]: v for i, v in
                           enumerate(mrp.get_value_function_vec(gamma))}\
            if matrix_method_for_mrp_eval else evaluate_mrp_result(mrp, gamma)
        improved_pi: FiniteDeterministicPolicy[S, A] = greedy_policy_from_vf(
            mdp,
            policy_vf,
            gamma
        )

        return policy_vf, improved_pi

    v_0: V[S] = {s: 0.0 for s in mdp.non_terminal_states}
    pi_0: FinitePolicy[S, A] = FinitePolicy(
        {s.state: Choose(mdp.actions(s)) for s in mdp.non_terminal_states}
    )
    return iterate(update, (v_0, pi_0))

def almost_equal_vf_pis(
    x1: Tuple[V[S], FinitePolicy[S, A]],
    x2: Tuple[V[S], FinitePolicy[S, A]]
) -> bool:
    return max(
        abs(x1[0][s] - x2[0][s]) for s in x1[0]
    ) < DEFAULT_TOLERANCE

def policy_iteration_result(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float,
) -> Tuple[V[S], FiniteDeterministicPolicy[S, A]]:
    return converged(policy_iteration(mdp, gamma), done=almost_equal_vf_pis)
```

If the number of non-terminal states of a given MDP is m and the number of actions ($|\mathcal{A}|$) is k, then the running time of Policy Improvement is $O(m^2 \cdot k)$ and we've already seen before that each iteration of Policy Evaluation is $O(m^2 \cdot k)$.

1.8 Bellman Optimality Operator and Value Iteration Algorithm

By making a small tweak to the definition of Greedy Policy Function in Equation (1.3) (changing the arg max to max), we define the *Bellman Optimality Operator*

$$\boldsymbol{B}^* : \mathbb{R}^m \to \mathbb{R}^m$$

as the following (non-linear) transformation of a vector (representing a Value Function) in the vector space $\mathbb{R}^m$

$$\boldsymbol{B}^*(\boldsymbol{V})(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{V}(s')\} \text{ for all } s \in \mathcal{N} \tag{1.5}$$

We shall use Equation (1.5) in our mathematical exposition but we require a different (but equivalent) expression for $\boldsymbol{B}^*(\boldsymbol{V})(s)$ to guide us with our code since the interface

for `FiniteMarkovDecisionProcess` operates on $\mathcal{P}_R$, rather than $\mathcal{R}$ and $\mathcal{P}$. The equivalent expression for $\boldsymbol{B}^*(\boldsymbol{V})(s)$ is as follows:

$$\boldsymbol{B}^*(\boldsymbol{V})(s) = \max_{a \in \mathcal{A}} \{\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{D}} \mathcal{P}_R(s, a, r, s') \cdot (r + \gamma \cdot \boldsymbol{W}(s'))\} \text{ for all } s \in \mathcal{N} \tag{1.6}$$

where $\boldsymbol{W} \in \mathbb{R}^n$ is defined (same as in the case of Equation (1.4)) as:

$$\boldsymbol{W}(s') = \begin{cases} \boldsymbol{V}(s') & \text{if } s' \in \mathcal{N} \\ 0 & \text{if } s' \in \mathcal{T} = \mathcal{S} - \mathcal{N} \end{cases}$$

Note that in Equation (1.6), because we have to work with $\mathcal{P}_R$, we need to consider transitions to all states $s' \in \mathcal{S}$ (versus transition to all states $s' \in \mathcal{N}$ in Equation (1.5)), and so, we need to handle the transitions to states $s' \in \mathcal{T}$ carefully (essentially by using the $\boldsymbol{W}$ function as described above).

For each $s \in \mathcal{N}$, the action $a \in \mathcal{A}$ that produces the maximization in (1.5) is the action prescribed by the deterministic policy π_D in (1.3). Therefore, if we apply the Bellman Policy Operator on any Value Function $\boldsymbol{V} \in \mathbb{R}^m$ using the Greedy Policy $G(\boldsymbol{V})$, it should be identical to applying the Bellman Optimality Operator. Therefore,

$$\boldsymbol{B}^{G(\boldsymbol{V})}(\boldsymbol{V}) = \boldsymbol{B}^*(\boldsymbol{V}) \text{ for all } \boldsymbol{V} \in \mathbb{R}^m \tag{1.7}$$

In particular, it's interesting to observe that by specializing $\boldsymbol{V}$ to be the Value Function $\boldsymbol{V}^\pi$ for a policy π, we get:

$$\boldsymbol{B}^{G(\boldsymbol{V}^\pi)}(\boldsymbol{V}^\pi) = \boldsymbol{B}^*(\boldsymbol{V}^\pi)$$

which is a succinct representation of the first stage of Policy Evaluation with an improved policy $G(\boldsymbol{V}^\pi)$ (note how all three of Bellman Policy Operator, Bellman Optimality Operator and Greedy Policy Function come together in this equation).

Much like how the Bellman Policy Operator $\boldsymbol{B}^\pi$ was motivated by the MDP Bellman Policy Equation (equivalently, the MRP Bellman Equation), Bellman Optimality Operator $\boldsymbol{B}^*$ is motivated by the MDP State-Value Function Bellman Optimality Equation (re-stated below):

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

Therefore, we can express the MDP State-Value Function Bellman Optimality Equation succinctly as:

$$\boldsymbol{V}^* = \boldsymbol{B}^*(\boldsymbol{V}^*)$$

which means $\boldsymbol{V}^* \in \mathbb{R}^m$ is a Fixed-Point of the Bellman Optimality Operator $\boldsymbol{B}^* : \mathbb{R}^m \to \mathbb{R}^m$.

Note that the definitions of the Greedy Policy Function and of the Bellman Optimality Operator that we have provided can be generalized to non-finite MDPs, and consequently we can generalize Equation (1.7) and the statement that $\boldsymbol{V}^*$ is a Fixed-Point of the Bellman Optimality Operator would still hold. However, in this chapter, since we are focused on developing algorithms for finite MDPs, we shall stick to the definitions we've provided for the case of finite MDPs.

Much like how we proved that $\boldsymbol{B}^{\pi}$ is a contraction function, we want to prove that $\boldsymbol{B}^{*}$ is a contraction function (under L^{∞} norm) so we can take advantage of Banach Fixed-Point Theorem and solve the Control problem by iterative applications of the Bellman Optimality Operator $\boldsymbol{B}^{*}$. So we need to prove that for all $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$,

$$\max_{s \in \mathcal{N}} |(\boldsymbol{B}^{*}(\boldsymbol{X}) - \boldsymbol{B}^{*}(\boldsymbol{Y}))(s)| \leq \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

This proof is a bit harder than the proof we did for $\boldsymbol{B}^{\pi}$. Here we need to utilize two key properties of $\boldsymbol{B}^{*}$.

1. Monotonicity Property, i.e, for all $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$,

   $$\text{If } \boldsymbol{X}(s) \geq \boldsymbol{Y}(s) \text{ for all } s \in \mathcal{N}, \text{ then } \boldsymbol{B}^{*}(\boldsymbol{X})(s) \geq \boldsymbol{B}^{*}(\boldsymbol{Y})(s) \text{ for all } s \in \mathcal{N}$$

   Observe that for each state $s \in \mathcal{N}$ and each action $a \in \mathcal{A}$,

   $$\{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{X}(s')\} - \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{Y}(s')\}$$

   $$= \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot (\boldsymbol{X}(s') - \boldsymbol{Y}(s')) \geq 0$$

   Therefore for each state $s \in \mathcal{N}$,

   $$\boldsymbol{B}^{*}(\boldsymbol{X})(s) - \boldsymbol{B}^{*}(\boldsymbol{Y})(s)$$

   $$= \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{X}(s')\} - \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{Y}(s')\} \geq 0$$

2. Constant Shift Property, i.e., for all $\boldsymbol{X} \in \mathbb{R}^m$, $c \in \mathbb{R}$,

   $$\boldsymbol{B}^{*}(\boldsymbol{X} + c)(s) = \boldsymbol{B}^{*}(\boldsymbol{X})(s) + \gamma c \text{ for all } s \in \mathcal{N}$$

   In the above statement, adding a constant ($\in \mathbb{R}$) to a Value Function ($\in \mathbb{R}^m$) adds the constant point-wise to all states of the Value Function (to all dimensions of the vector representing the Value Function). In other words, a constant $\in \mathbb{R}$ might as well be treated as a Value Function with the same (constant) value for all states. Therefore,

   $$\boldsymbol{B}^{*}(\boldsymbol{X} + c)(s) = \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot (\boldsymbol{X}(s') + c)\}$$

   $$= \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot \boldsymbol{X}(s')\} + \gamma c = \boldsymbol{B}^{*}(\boldsymbol{X})(s) + \gamma c$$

With these two properties of $\boldsymbol{B}^{*}$ in place, let's prove that $\boldsymbol{B}^{*}$ is a contraction function. For given $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^m$, assume:

$$\max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)| = c$$

We can rewrite this as:

$$\boldsymbol{X}(s) - c \leq \boldsymbol{Y}(s) \leq \boldsymbol{X}(s) + c \text{ for all } s \in \mathcal{N}$$

Since $\boldsymbol{B}^*$ has the monotonicity property, we can apply $\boldsymbol{B}^*$ throughout the above double-inequality.

$$\boldsymbol{B}^*(\boldsymbol{X} - c)(s) \leq \boldsymbol{B}^*(\boldsymbol{Y})(s) \leq \boldsymbol{B}^*(\boldsymbol{X} + c)(s) \text{ for all } s \in \mathcal{N}$$

Since $\boldsymbol{B}^*$ has the constant shift property,

$$\boldsymbol{B}^*(\boldsymbol{X})(s) - \gamma c \leq \boldsymbol{B}^*(\boldsymbol{Y})(s) \leq \boldsymbol{B}^*(\boldsymbol{X})(s) + \gamma c \text{ for all } s \in \mathcal{N}$$

In other words,

$$\max_{s \in \mathcal{N}} |(\boldsymbol{B}^*(\boldsymbol{X}) - \boldsymbol{B}^*(\boldsymbol{Y}))(s)| \leq \gamma c = \gamma \cdot \max_{s \in \mathcal{N}} |(\boldsymbol{X} - \boldsymbol{Y})(s)|$$

So invoking Banach Fixed-Point Theorem proves the following Theorem:

Theorem 1.8.1 (Value Iteration Convergence Theorem)**.** *For a Finite MDP with $|\mathcal{N}| = m$ and $\gamma < 1$, if $\boldsymbol{V}^* \in \mathbb{R}^m$ is the Optimal Value Function, then $\boldsymbol{V}^*$ is the unique Fixed-Point of the Bellman Optimality Operator $\boldsymbol{B}^* : \mathbb{R}^m \to \mathbb{R}^m$, and*

$$\lim_{i \to \infty} (\boldsymbol{B}^*)^i(\boldsymbol{V_0}) \to \boldsymbol{V}^* \text{ for all starting Value Functions } \boldsymbol{V_0} \in \mathbb{R}^m$$

This gives us the following iterative algorithm, known as the *Value Iteration* algorithm, due to Richard Bellman (Bellman 1957a):

- Start with any Value Function $\boldsymbol{V_0} \in \mathbb{R}^m$
- Iterating over $i = 0, 1, 2, \ldots$, calculate in each iteration:

$$\boldsymbol{V_{i+1}}(s) = \boldsymbol{B}^*(\boldsymbol{V_i})(s) \text{ for all } s \in \mathcal{N}$$

We stop the algorithm when $d(\boldsymbol{V_i}, \boldsymbol{V_{i+1}}) = \max_{s \in \mathcal{N}} |(\boldsymbol{V_i} - \boldsymbol{V_{i+1}})(s)|$ is adequately small.

It pays to emphasize that Banach Fixed-Point Theorem not only assures convergence to the unique solution $\boldsymbol{V}^*$ (no matter what Value Function $\boldsymbol{V_0}$ we start the algorithm with), it also assures a reasonable speed of convergence (dependent on the choice of starting Value Function $\boldsymbol{V_0}$ and the choice of γ).

1.9 Optimal Policy from Optimal Value Function

Note that the Policy Iteration algorithm produces a policy together with a Value Function in each iteration. So, in the end, when we converge to the Optimal Value Function $\boldsymbol{V_j} = \boldsymbol{V}^*$ in iteration j, the Policy Iteration algorithm has a deterministic policy π_j associated with $\boldsymbol{V_j}$ such that:

$$\boldsymbol{V_j} = \boldsymbol{V}^{\pi_j} = \boldsymbol{V}^*$$

and we refer to π_j as the Optimal Policy π^*, one that yields the Optimal Value Function $\boldsymbol{V}^*$, i.e.,

$$\boldsymbol{V}^{\pi^*} = \boldsymbol{V}^*$$

But Value Iteration has no such policy associated with it since the entire algorithm is devoid of a policy representation and operates only with Value Functions. So now the question is: when Value Iteration converges to the Optimal Value Function $\boldsymbol{V_i} = \boldsymbol{V}^*$ in iteration i, how do we get hold of an Optimal Policy π^* such that:

$$\boldsymbol{V}^{\pi^*} = \boldsymbol{V_i} = \boldsymbol{V}^*$$

The answer lies in the Greedy Policy function G. Equation (1.7) told us that:

$$\boldsymbol{B}^{G(\boldsymbol{V})}(\boldsymbol{V}) = \boldsymbol{B}^*(\boldsymbol{V}) \text{ for all } \boldsymbol{V} \in \mathbb{R}^m$$

Specializing $\boldsymbol{V}$ to be $\boldsymbol{V}^*$, we get:

$$\boldsymbol{B}^{G(\boldsymbol{V}^*)}(\boldsymbol{V}^*) = \boldsymbol{B}^*(\boldsymbol{V}^*)$$

But we know that $\boldsymbol{V}^*$ is the Fixed-Point of the Bellman Optimality Operator $\boldsymbol{B}^*$, i.e., $\boldsymbol{B}^*(\boldsymbol{V}^*) = \boldsymbol{V}^*$. Therefore,

$$\boldsymbol{B}^{G(\boldsymbol{V}^*)}(\boldsymbol{V}^*) = \boldsymbol{V}^*$$

The above equation says $\boldsymbol{V}^*$ is the Fixed-Point of the Bellman Policy Operator $\boldsymbol{B}^{G(\boldsymbol{V}^*)}$. However, we know that $\boldsymbol{B}^{G(\boldsymbol{V}^*)}$ has a unique Fixed-Point equal to $\boldsymbol{V}^{G(\boldsymbol{V}^*)}$. Therefore,

$$\boldsymbol{V}^{G(\boldsymbol{V}^*)} = \boldsymbol{V}^*$$

This says that evaluating the MDP with the deterministic greedy policy $G(\boldsymbol{V}^*)$ (policy created from the Optimal Value Function $\boldsymbol{V}^*$ using the Greedy Policy Function G) in fact achieves the Optimal Value Function $\boldsymbol{V}^*$. In other words, $G(\boldsymbol{V}^*)$ is the (Deterministic) Optimal Policy π^* we've been seeking.

Now let's write the code for Value Iteration. The function `value_iteration` returns an `Iterator` on Value Functions (of type `V[S]`) produced by the Value Iteration algorithm. It uses the function `update` for application of the Bellman Optimality Operator. `update` prepares the Q-Values for a state by looping through all the allowable actions for the state, and then calculates the maximum of those Q-Values (over the actions). The Q-Value calculation is same as what we saw in `greedy_policy_from_vf`: $\mathbb{E}_{(s',r)\sim\mathcal{P}_R}[r + \gamma \cdot \boldsymbol{W}(s')]$, using the $\mathcal{P}_R$ probabilities represented in the `mapping` attribute of the `mdp` object (essentially Equation (1.6)). Note the use of the previously-written function `extended_vf` to handle the function $\boldsymbol{W} : \mathcal{S} \to \mathbb{R}$ that appears in the definition of Bellman Optimality Operator in Equation (1.6). The function `value_iteration_result` returns the final (optimal) Value Function, together with it's associated Optimal Policy. It simply returns the last Value Function of the `Iterator[V[S]]` returned by `value_iteration`, using the termination condition specified in `almost_equal_vfs`.

```
DEFAULT_TOLERANCE = 1e-5

def value_iteration(
    mdp: FiniteMarkovDecisionProcess[S, A],
    gamma: float
) -> Iterator[V[S]]:
    def update(v: V[S]) -> V[S]:
        return {s: max(mdp.mapping[s][a].expectation(
            lambda s_r: s_r[1] + gamma * extended_vf(v, s_r[0])
        ) for a in mdp.actions(s)) for s in v}

    v_0: V[S] = {s: 0.0 for s in mdp.non_terminal_states}
    return iterate(update, v_0)

def almost_equal_vfs(
    v1: V[S],
    v2: V[S],
    tolerance: float = DEFAULT_TOLERANCE
) -> bool:
    return max(abs(v1[s] - v2[s]) for s in v1) < tolerance

def value_iteration_result(
    mdp: FiniteMarkovDecisionProcess[S, A],
```

```
    gamma: float
) -> Tuple[V[S], FiniteDeterministicPolicy[S, A]]:
    opt_vf: V[S] = converged(
        value_iteration(mdp, gamma),
        done=almost_equal_vfs
    )
    opt_policy: FiniteDeterministicPolicy[S, A] = greedy_policy_from_vf(
        mdp,
        opt_vf,
        gamma
    )
    return opt_vf, opt_policy
```

If the number of non-terminal states of a given MDP is m and the number of actions ($|\mathcal{A}|$) is k, then the running time of each iteration of Value Iteration is $O(m^2 \cdot k)$.

We encourage you to play with the above implementations of Policy Evaluation, Policy Iteration and Value Iteration (code in the file rl/dynamic_programming.py) by running it on MDPs/Policies of your choice, and observing the traces of the algorithms.

1.10 Revisiting the Simple Inventory Example

Let's revisit the simple inventory example. We shall consider the version with a space capacity since we want an example of a `FiniteMarkovDecisionProcess`. It will help us test our code for Policy Evaluation, Policy Iteration and Value Iteration. More importantly, it will help us identify the mathematical structure of the optimal policy of ordering for this store inventory problem. So let's take another look at the code we wrote in Chapter **??** to set up an instance of a `SimpleInventoryMDPCap` and a `FiniteDeterministicPolicy` (that we can use for Policy Evaluation).

```
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)}
)
```

Now let's write some code to evaluate `si_mdp` with the policy `fdp`.

```
from pprint import pprint
implied_mrp: FiniteMarkovRewardProcess[InventoryState] =\
    si_mdp.apply_finite_policy(fdp)
user_gamma = 0.9
pprint(evaluate_mrp_result(implied_mrp, gamma=user_gamma))
```

This prints the following Value Function.

```
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -35.510518165628724,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.93217421014731,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -28.345029758390766,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.93217421014731,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -29.345029758390766,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.345029758390766}
```

Next, let's run Policy Iteration.

```
opt_vf_pi, opt_policy_pi = policy_iteration_result(
    si_mdp,
    gamma=user_gamma
)
pprint(opt_vf_pi)
print(opt_policy_pi)
```

This prints the following Optimal Value Function and Optimal Policy.

```
{NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.660960231637507,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -27.991900091403533,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.660960231637507,
 NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -34.894855781630035,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -28.991900091403533,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -29.991900091403533}

For State InventoryState(on_hand=0, on_order=0): Do Action 1
For State InventoryState(on_hand=0, on_order=1): Do Action 1
For State InventoryState(on_hand=0, on_order=2): Do Action 0
For State InventoryState(on_hand=1, on_order=0): Do Action 1
For State InventoryState(on_hand=1, on_order=1): Do Action 0
For State InventoryState(on_hand=2, on_order=0): Do Action 0
```

As we can see, the Optimal Policy is to not order if the Inventory Position (sum of On-Hand and On-Order) is greater than 1 unit and to order 1 unit if the Inventory Position is 0 or 1. Finally, let's run Value Iteration.

```
opt_vf_vi, opt_policy_vi = value_iteration_result(si_mdp, gamma=user_gamma
)
pprint(opt_vf_vi)
print(opt_policy_vi)
```

You'll see the output from Value Iteration matches the output produced from Policy Iteration - this is a good validation of our code correctness. We encourage you to play around with `user_capacity`, `user_poisson_lambda`, `user_holding_cost`, `user_stockout_cost` and `user_gamma`(code in `__main__` in rl/chapter3/simple_inventory_mdp_cap.py). As a valuable exercise, using this code, discover the mathematical structure of the Optimal Policy as a function of the above inputs.

1.11 Generalized Policy Iteration

In this section, we dig into the structure of the Policy Iteration algorithm and show how this structure can be generalized. Let us start by looking at a 2-dimensional layout of how

the Value Functions progress in Policy Iteration from the starting Value Function $\boldsymbol{V_0}$ to the final Value Function $\boldsymbol{V}^*$.

$$\pi_1 = G(\boldsymbol{V_0}), \boldsymbol{V_0} \rightarrow \boldsymbol{B}^{\pi_1}(\boldsymbol{V_0}) \rightarrow (\boldsymbol{B}^{\pi_1})^2(\boldsymbol{V_0}) \rightarrow \ldots (\boldsymbol{B}^{\pi_1})^i(\boldsymbol{V_0}) \rightarrow \ldots \boldsymbol{V}^{\pi_1} = \boldsymbol{V_1}$$

$$\pi_2 = G(\boldsymbol{V_1}), \boldsymbol{V_1} \rightarrow \boldsymbol{B}^{\pi_2}(\boldsymbol{V_1}) \rightarrow (\boldsymbol{B}^{\pi_2})^2(\boldsymbol{V_1}) \rightarrow \ldots (\boldsymbol{B}^{\pi_2})^i(\boldsymbol{V_1}) \rightarrow \ldots \boldsymbol{V}^{\pi_2} = \boldsymbol{V_2}$$

$$\ldots$$

$$\ldots$$

$$\pi_{j+1} = G(\boldsymbol{V_j}), \boldsymbol{V_j} \rightarrow \boldsymbol{B}^{\pi_{j+1}}(\boldsymbol{V_j}) \rightarrow (\boldsymbol{B}^{\pi_{j+1}})^2(\boldsymbol{V_j}) \rightarrow \ldots (\boldsymbol{B}^{\pi_{j+1}})^i(\boldsymbol{V_j}) \rightarrow \ldots \boldsymbol{V}^{\pi_{j+1}} = \boldsymbol{V}^*$$

Each row in the layout above represents the progression of the Value Function for a specific policy. Each row starts with the creation of the policy (for that row) using the Greedy Policy Function G, and the remainder of the row consists of successive applications of the Bellman Policy Operator (using that row's policy) until convergence to the Value Function for that row's policy. So each row starts with a Policy Improvement and the rest of the row is a Policy Evaluation. Notice how the end of one row dovetails into the start of the next row with application of the Greedy Policy Function G. It's also important to recognize that Greedy Policy Function as well as Bellman Policy Operator apply to *all states* in $\mathcal{N}$. So, in fact, the entire Policy Iteration algorithm has 3 nested loops. The outermost loop is over the rows in this 2-dimensional layout (each iteration in this outermost loop creates an improved policy). The loop within this outermost loop is over the columns in each row (each iteration in this loop applies the Bellman Policy Operator, i.e. the iterations of Policy Evaluation). The innermost loop is over each state in $\mathcal{N}$ since we need to sweep through all states in updating the Value Function when the Bellman Policy Operator is applied on a Value Function (we also need to sweep through all states in applying the Greedy Policy Function to improve the policy).

A higher-level view of Policy Iteration is to think of Policy Evaluation and Policy Improvement going back and forth iteratively - Policy Evaluation takes a policy and creates the Value Function for that policy, while Policy Improvement takes a Value Function and creates a Greedy Policy from it (that is improved relative to the previous policy). This was depicted in Figure 1.1. It is important to recognize that this loop of Policy Evaluation and Policy Improvement works to make the Value Function and the Policy increasingly consistent with each other, until we reach convergence when the Value Function and Policy become completely consistent with each other (as was illustrated in Figure 1.2).

We'd also like to share a visual of Policy Iteration that is quite popular in much of the literature on Dynamic Programming, originally appearing in Sutton and Barto's RL book (Sutton and Barto 2018). It is the visual of Figure 1.3. It's a somewhat fuzzy sort of visual, but it has it's benefits in terms of pedagogy of Policy Iteration. The idea behind this image is that the lower line represents the "policy line" indicating the progression of the policies as Policy Iteration algorithm moves along and the upper line represents the "value function line" indicating the progression of the Value Functions as Policy Iteration algorithm moves along. The arrows pointing towards the upper line ("value function line") represent a Policy Evaluation for a given policy π, yielding the point (Value Function) $\boldsymbol{V}^\pi$ on the upper line. The arrows pointing towards the lower line ("policy line") represent a Greedy Policy Improvement from a Value Function $\boldsymbol{V}^\pi$, yielding the point (policy) $\pi' = G(\boldsymbol{V}^\pi)$ on the lower line. The key concept here is that Policy Evaluation (arrows pointing to upper line) and Policy Improvement (arrows pointing to lower line) are "competing" - they "push in different directions" even as they aim to get the Value Function and Policy to be

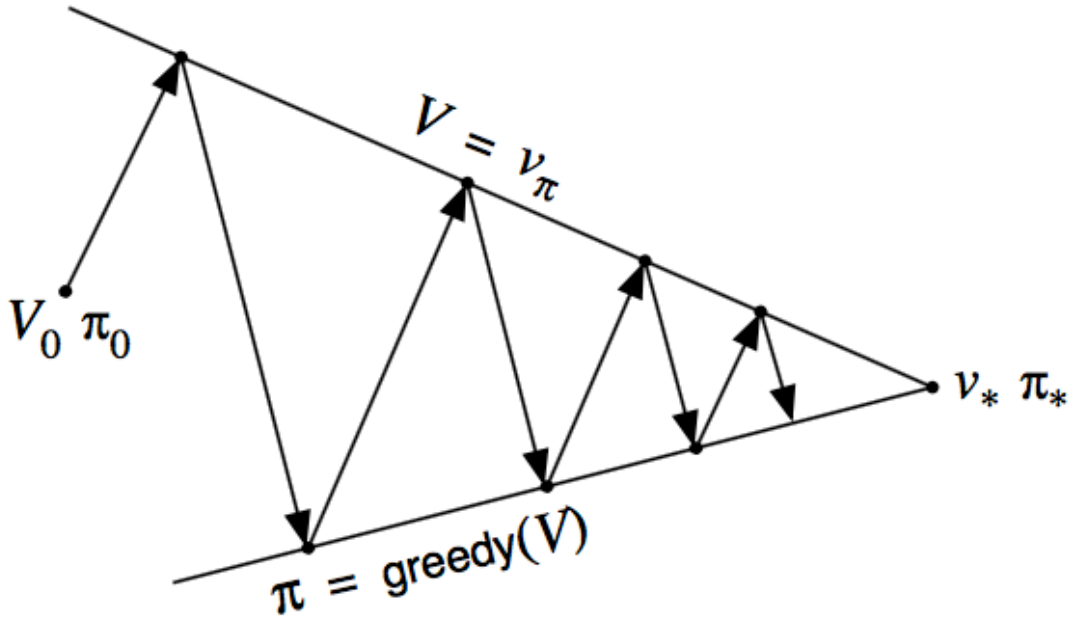

Figure 1.3: Progression Lines of Value Function and Policy in Policy Iteration (Image Credit: Sutton-Barto's RL Book)

consistent with each other. This concept of simultaneously trying to compete and trying to be consistent might seem confusing and contradictory, so it deserves a proper explanation. Things become clear by noting that there are actually two notions of consistency between a Value Function $\boldsymbol{V}$ and Policy π.

1. The notion of the Value Function $\boldsymbol{V}$ being consistent with/close to the Value Function $\boldsymbol{V}^{\pi}$ of the policy π.
2. The notion of the Policy π being consistent with/close to the Greedy Policy $G(\boldsymbol{V})$ of the Value Function $\boldsymbol{V}$.

Policy Evaluation aims for the first notion of consistency, but in the process, makes it worse in terms of the second notion of consistency. Policy Improvement aims for the second notion of consistency, but in the process, makes it worse in terms of the first notion of consistency. This also helps us understand the rationale for alternating between Policy Evaluation and Policy Improvement so that neither of the above two notions of consistency slip up too much (thanks to the alternating propping up of the two notions of consistency). Also, note that as Policy Iteration progresses, the upper line and lower line get closer and closer and the "pushing in different directions" looks more and more collaborative rather than competing (the gaps in consistency become lesser and lesser). In the end, the two lines intersect, when there is no more pushing to do for either of Policy Evaluation or Policy Improvement since at convergence, π^* and $\boldsymbol{V}^*$ have become completely consistent.

Now we are ready to talk about a very important idea known as *Generalized Policy Iteration* that is emphasized throughout Sutton and Barto's RL book (Sutton and Barto 2018) as the perspective that unifies all variants of DP as well as RL algorithms. Generalized Policy Iteration is the idea that we can evaluate the Value Function for a policy with *any*

Policy Evaluation method, and we can improve a policy with *any* Policy Improvement method (not necessarily the methods used in the classical Policy Iteration DP algorithm). In particular, we'd like to emphasize the idea that neither of Policy Evaluation and Policy Improvement need to go fully towards the notion of consistency they are respectively striving for. As a simple example, think of modifying Policy Evaluation (say for a policy π) to not go all the way to $\boldsymbol{V}^{\pi}$, but instead just perform say 3 Bellman Policy Evaluations. This means it would partially bridge the gap on the first notion of consistency (getting closer to $\boldsymbol{V}^{\pi}$ but not go all the way to $\boldsymbol{V}^{\pi}$), but it would also mean not slipping up too much on the second notion of consistency. As another example, think of updating just 5 of the states (say in a large state space) with the Greedy Policy Improvement function (rather than the normal Greedy Policy Improvement function that operates on all the states). This means it would partially bridge the gap on the second notion of consistency (getting closer to $G(\boldsymbol{V}^{\pi})$ but not go all the way to $G(\boldsymbol{V}^{\pi})$), but it would also mean not slipping up too much on the first notion of consistency. A concrete example of Generalized Policy Iteration is in fact Value Iteration. In Value Iteration, we apply the Bellman Policy Operator just once before moving on to Policy Improvement. In a 2-dimensional layout, this is what Value Iteration looks like:

$$\pi_1 = G(\boldsymbol{V_0}), \boldsymbol{V_0} \rightarrow \boldsymbol{B}^{\pi_1}(\boldsymbol{V_0}) = \boldsymbol{V_1}$$

$$\pi_2 = G(\boldsymbol{V_1}), \boldsymbol{V_1} \rightarrow \boldsymbol{B}^{\pi_2}(\boldsymbol{V_1}) = \boldsymbol{V_2}$$

$$\ldots$$

$$\ldots$$

$$\pi_{j+1} = G(\boldsymbol{V_j}), \boldsymbol{V_j} \rightarrow \boldsymbol{B}^{\pi_{j+1}}(\boldsymbol{V_j}) = \boldsymbol{V}^*$$

So the greedy policy improvement step is unchanged, but Policy Evaluation is reduced to just a single Bellman Policy Operator application. In fact, pretty much all control algorithms in Reinforcement Learning can be viewed as special cases of Generalized Policy Iteration. In some of the simple versions of Reinforcement Learning Control algorithms, the Policy Evaluation step is done for just a single state (versus for all states in usual Policy Iteration, or even in Value Iteration) and the Policy Improvement step is also done for just a single state. So essentially these Reinforcement Learning Control algorithms are an alternating sequence of single-state policy evaluation and single-state policy improvement (where the single-state is the state produced by sampling or the state that is encountered in a real-world environment interaction). Figure 1.4 illustrates Generalized Policy Iteration as the shorter-length arrows (versus the longer-length arrows seen in Figure 1.3 for the usual Policy Iteration algorithm). Note how these shorter-length arrows don't go all the way to either the "value function line" or the "policy line" but they do go some part of the way towards the line they are meant to go towards at that stage in the algorithm.

We would go so far as to say that the Bellman Equations and the concept of Generalized Policy Iteration are the two most important concepts to internalize in the study of Reinforcement Learning, and we highly encourage you to think along the lines of these two ideas when we present several algorithms later in this book. The importance of the concept of Generalized Policy Iteration (GPI) might not be fully visible to you yet, but we hope that GPI will be your mantra by the time you finish this book. For now, let's just note the key takeaway regarding GPI - it is any algorithm to solve MDP control that alternates between *some form of* Policy Evaluation and *some form of* Policy Improvement. We will bring up GPI several times later in this book.

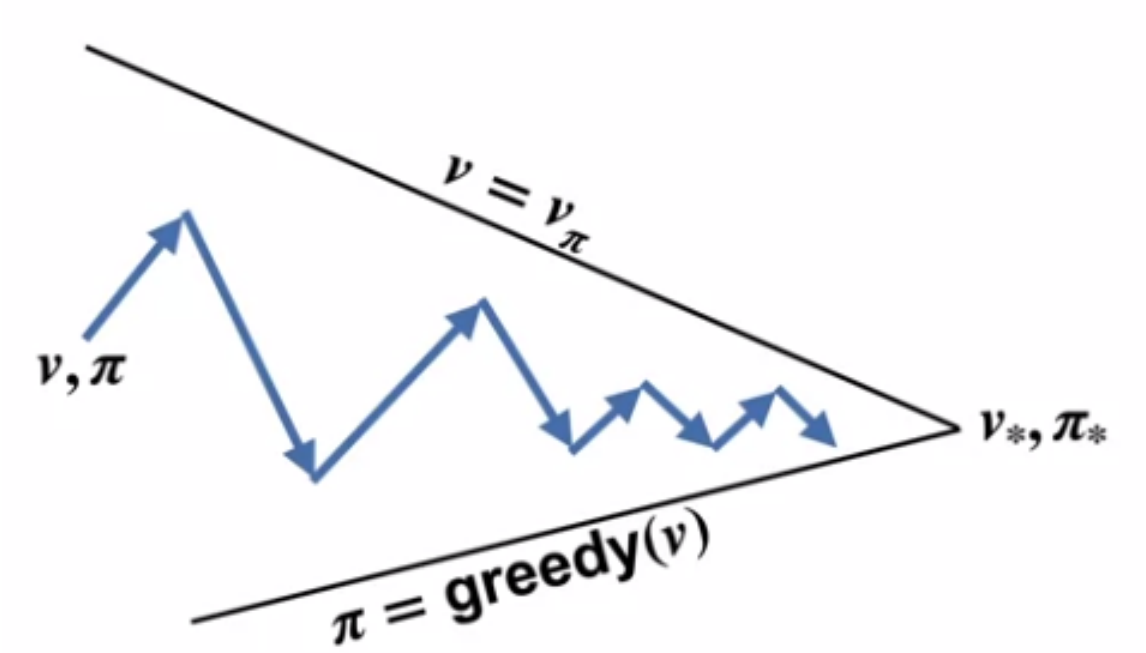

Figure 1.4: Progression Lines of Value Function and Policy in Generalized Policy Iteration (Image Credit: Coursera Course on Fundamentals of RL)

1.12 Aysnchronous Dynamic Programming

The classical Dynamic Programming algorithms we have described in this chapter are qualified as *Synchronous* Dynamic Programming algorithms. The word *synchronous* refers to two things:

1. All states' values are updated in each iteration
2. The mathematical description of the algorithms corresponds to all the states' value updates to occur simultaneously. However, when implementing in code (in Python, where computation is serial and not parallel), this "simultaneous update" would be done by creating a new copy of the Value Function vector and sweeping through all states to assign values to the new copy from the values in the old copy.

In practice, Dynamic Programming algorithms are typically implemented as *Asynchronous* algorithms, where the above two constraints (all states updated simultaneously) are relaxed. The term *asynchronous* affords a lot of flexibility - we can update a subset of states in each iteration, and we can update states in any order we like. A natural outcome of this relaxation of the synchronous constraint is that we can maintain just one vector for the value function and update the values *in-place*. This has considerable benefits - an updated value for a state is immediately available for updates of other states (note: in synchronous, with the old and new value function vectors, one has to wait for the entire states sweep to be over until an updated state value is available for another state's update). In fact, in-place updates of value function is the norm in practical implementations of algorithms to solve the MDP Control problem.

Another feature of practical asynchronous algorithms is that we can prioritize the order in which state values are updated. There are many ways in which algorithms assign prior-

ities, and we'll just highlight a simple but effective way of prioritizing state value updates. It's known as *prioritized sweeping*. We maintain a queue of the states, sorted by their "value function gaps" $g : \mathcal{N} \to \mathbb{R}$ (illustrated below as an example for Value Iteration):

$$g(s) = |V(s) - \max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V(s')\}| \text{ for all } s \in \mathcal{N}$$

After each state's value is updated with the Bellman Optimality Operator, we update the Value Function Gap for all the states whose Value Function Gap does get changed as a result of this state value update. These are exactly the states from which we have a probabilistic transition to the state whose value just got updated. What this also means is that we need to maintain the reverse transition dynamics in our data structure representation. So, after each state value update, the queue of states is resorted (by their value function gaps). We always pull out the state with the largest value function gap (from the top of the queue), and update the value function for that state. This prioritizes updates of states with the largest gaps, and it ensures that we quickly get to a point where all value function gaps are low enough.

Another form of Asynchronous Dynamic Programming worth mentioning here is *Real-Time Dynamic Programming* (RTDP). RTDP means we run a Dynamic Programming algorithm *while* the AI agent is experiencing real-time interaction with the environment. When a state is visited during the real-time interaction, we make an update for that state's value. Then, as we transition to another state as a result of the real-time interaction, we update that new state's value, and so on. Note also that in RTDP, the choice of action is the real-time action executed by the AI agent, which the environment responds to. This action choice is governed by the policy implied by the value function for the encountered state at that point in time in the real-time interaction.

Finally, we need to highlight that often special types of structures of MDPs can benefit from specific customizations of Dynamic Programming algorithms (typically, Asynchronous). One such specialization is when each state is encountered not more than once in each random sequence of state occurrences when an AI agent plays out an MDP, and when all such random sequences of the MDP terminate. This structure can be conceptualized as a Directed Acylic Graph wherein each non-terminal node in the Directed Acyclic Graph (DAG) represents a pair of non-terminal state and action, and each terminal node in the DAG represents a terminal state (the graph edges represent probabilistic transitions of the MDP). In this specialization, the MDP Prediction and Control problems can be solved in a fairly simple manner - by walking backwards on the DAG from the terminal nodes and setting the Value Function of visited states (in the backward DAG walk) using the Bellman Optimality Equation (for Control) or Bellman Policy Equation (for Prediction). Here we don't need the "iterate to convergence" approach of Policy Evaluation or Policy Iteration or Value Iteration. Rather, all these Dynamic Programming algorithms essentially reduce to a simple back-propagation of the Value Function on the DAG. This means, states are visited (and their Value Functions set) in the order determined by the reverse sequence of a Topological Sort on the DAG. We shall make this DAG back-propagation Dynamic Programming algorithm clear for a special DAG structure - Finite-Horizon MDPs - where all random sequences of the MDP terminate within a fixed number of time steps and each time step has a separate (from other time steps) set of states. This special case of Finite-Horizon MDPs is fairly common in Financial Applications and so, we cover it in detail in the next section.

1.13 Finite-Horizon Dynamic Programming: Backward Induction

In this section, we consider a specialization of the DAG-structured MDPs described at the end of the previous section - one that we shall refer to as *Finite-Horizon MDPs*, where each sequence terminates within a fixed finite number of time steps T and each time step has a separate (from other time steps) set of countable states. So, all states at time-step T are terminal states and some states before time-step T could be terminal states. For all $t = 0, 1, \ldots, T$, denote the set of states for time step t as $\mathcal{S}_t$, the set of terminal states for time step t as $\mathcal{T}_t$ and the set of non-terminal states for time step t as $\mathcal{N}_t = \mathcal{S}_t - \mathcal{T}_t$ (note: $\mathcal{N}_T = \emptyset$). As mentioned previously, when the MDP is not time-homogeneous, we augment each state to include the index of the time step so that the augmented state at time step t is (t, s_t) for $s_t \in \mathcal{S}_t$. The entire MDP's (augmented) state space $\mathcal{S}$ is:

$$\{(t, s_t) | t = 0, 1, \ldots, T, s_t \in \mathcal{S}_t\}$$

We need a Python class to represent this augmented state space.

```
@dataclass(frozen=True)
class WithTime(Generic[S]):
    state: S
    time: int = 0
```

The set of terminal states $\mathcal{T}$ is:

$$\{(t, s_t) | t = 0, 1, \ldots, T, s_t \in \mathcal{T}_t\}$$

As usual, the set of non-terminal states is denoted as $\mathcal{N} = \mathcal{S} - \mathcal{T}$.

We denote the set of rewards receivable by the AI agent at time t as $\mathcal{D}_t$ (countable subset of $\mathbb{R}$) and we denote the allowable actions for states in $\mathcal{N}_t$ as $\mathcal{A}_t$. In a more generic setting, as we shall represent in our code, each non-terminal state (t, s_t) has it's own set of allowable actions, denoted $\mathcal{A}(s_t)$, However, for ease of exposition, here we shall treat all non-terminal states at a particular time step to have the same set of allowable actions $\mathcal{A}_t$. Let us denote the entire action space $\mathcal{A}$ of the MDP as the union of all the $\mathcal{A}_t$ over all $t = 0, 1, \ldots, T - 1$.

The state-reward transition probability function

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

is given by:

$$\mathcal{P}_R((t, s_t), a_t, r_{t'}, (t', s_{t'})) = \begin{cases} (\mathcal{P}_R)_t(s_t, a_t, r_{t'}, s_{t'}) & \text{if } t' = t + 1 \text{ and } s_{t'} \in \mathcal{S}_{t'} \text{ and } r_{t'} \in \mathcal{D}_{t'} \\ 0 & \text{otherwise} \end{cases}$$

for all $t = 0, 1, \ldots T - 1, s_t \in \mathcal{N}_t, a_t \in \mathcal{A}_t, t' = 0, 1, \ldots, T$ where

$$(\mathcal{P}_R)_t : \mathcal{N}_t \times \mathcal{A}_t \times \mathcal{D}_{t+1} \times \mathcal{S}_{t+1} \to [0, 1]$$

are the separate state-reward transition probability functions for each of the time steps $t = 0, 1, \ldots, T - 1$ such that

$$\sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r_{t+1} \in \mathcal{D}_{t+1}} (\mathcal{P}_R)_t(s_t, a_t, r_{t+1}, s_{t+1}) = 1$$

for all $t = 0, 1, \ldots, T - 1, s_t \in \mathcal{N}_t, a_t \in \mathcal{A}_t$.

So it is convenient to represent a finite-horizon MDP with separate state-reward transition probability functions $(\mathcal{P}_R)_t$ for each time step. Likewise, it is convenient to represent any policy of the MDP

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

as:

$$\pi((t, s_t), a_t) = \pi_t(s_t, a_t)$$

where

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

are the separate policies for each of the time steps $t = 0, 1, \ldots, T-1$

So essentially we interpret π as being composed of the sequence $(\pi_0, \pi_1, \ldots, \pi_{T-1})$.

Consequently, the Value Function for a given policy π (equivalently, the Value Function for the π-implied MRP)

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

can be conveniently represented in terms of a sequence of Value Functions

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

for each of time steps $t = 0, 1, \ldots, T-1$, defined as:

$$V^\pi((t, s_t)) = V_t^\pi(s_t) \text{ for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t$$

Then, the Bellman Policy Equation can be written as:

$$V_t^\pi(s_t) = \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r_{t+1} \in \mathcal{D}_{t+1}} (\mathcal{P}_R^{\pi_t})_t(s_t, r_{t+1}, s_{t+1}) \cdot (r_{t+1} + \gamma \cdot W_{t+1}^\pi(s_{t+1})) \quad \text{for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t \tag{1.8}$$

where

$$W_t^\pi(s_t) = \begin{cases} V_t^\pi(s_t) & \text{if } s_t \in \mathcal{N}_t \\ 0 & \text{if } s_t \in \mathcal{T}_t \end{cases} \text{ for all } t = 1, 2, \ldots, T$$

and where $(\mathcal{P}_R^{\pi_t})_t : \mathcal{N}_t \times \mathcal{D}_{t+1} \times \mathcal{S}_{t+1}$ for all $t = 0, 1, \ldots, T-1$ represent the π-implied MRP's state-reward transition probability functions for the time steps, defined as:

$$(\mathcal{P}_R^{\pi_t})_t(s_t, r_{t+1}, s_{t+1}) = \sum_{a_t \in \mathcal{A}_t} \pi_t(s_t, a_t) \cdot (\mathcal{P}_R)_t(s_t, a_t, r_{t+1}, s_{t+1}) \text{ for all } t = 0, 1, \ldots, T-1$$

So for a Finite MDP, this yields a simple algorithm to calculate V_t^π for all t by simply decrementing down from $t = T-1$ to $t = 0$ and using Equation (1.8) to calculate V_t^π for all $t = 0, 1, \ldots, T-1$ from the known values of W_{t+1}^π (since we are decrementing in time index t).

This algorithm is the adaptation of Policy Evaluation to the finite horizon case with this simple technique of "stepping back in time" (known as *Backward Induction*). Let's write

some code to implement this algorithm. We are given an MDP over the augmented (finite) state space `WithTime[S]`, and a policy π (also over the augmented state space `WithTime[S]`). So, we can use the method `apply_finite_policy` in `FiniteMarkovDecisionProcess[WithTime[S], A]` to obtain the π-implied MRP of type `FiniteMarkovRewardProcess[WithTime[S]]`.

Our first task to to "unwrap" the state-reward probability transition function $\mathcal{P}_R^\pi$ of this π-implied MRP into a time-indexed sequenced of state-reward probability transition functions $(\mathcal{P}_R^{\pi_t})_t, t = 0, 1, \ldots, T-1$. This is accomplished by the following function `unwrap_finite_horizon_MRP` (`itertools.groupby` groups the augmented states by their time step, and the function `without_time` strips the time step from the augmented states when placing the states in $(\mathcal{P}_R^{\pi_t})_t$, i.e., `Sequence[RewardTransition[S]]`).

```
from itertools import groupby

StateReward = FiniteDistribution[Tuple[State[S], float]]
RewardTransition = Mapping[NonTerminal[S], StateReward[S]]

def unwrap_finite_horizon_MRP(
    process: FiniteMarkovRewardProcess[WithTime[S]]
) -> Sequence[RewardTransition[S]]:

    def time(x: WithTime[S]) -> int:
        return x.time

    def single_without_time(
        s_r: Tuple[State[WithTime[S]], float]
    ) -> Tuple[State[S], float]:
        if isinstance(s_r[0], NonTerminal):
            ret: Tuple[State[S], float] = (
                NonTerminal(s_r[0].state.state),
                s_r[1]
            )
        else:
            ret = (Terminal(s_r[0].state.state), s_r[1])
        return ret

    def without_time(arg: StateReward[WithTime[S]]) -> StateReward[S]:
        return arg.map(single_without_time)

    return [{NonTerminal(s.state): without_time(
        process.transition_reward(NonTerminal(s))
    ) for s in states} for _, states in groupby(
        sorted(
            (nt.state for nt in process.non_terminal_states),
            key=time
        ),
        key=time
    )]
```

Now that we have the state-reward transition functions $(\mathcal{P}_R^{\pi_t})_t$ arranged in the form of a `Sequence[RewardTransition[S]]`, we are ready to perform backward induction to calculate V_t^π. The following function `evaluate` accomplishes it with a straightforward use of Equation (1.8), as described above. Note the use of the previously-written `extended_vf` function, that represents the $W_t^\pi : \mathcal{S}_t \to \mathbb{R}$ function appearing on the right-hand-side of Equation (1.8).

```
def evaluate(
    steps: Sequence[RewardTransition[S]],
    gamma: float
) -> Iterator[V[S]]:
    v: List[V[S]] = []

    for step in reversed(steps):
        v.append({s: res.expectation(
```

```
                lambda s_r: s_r[1] + gamma * (
                    extended_vf(v[-1], s_r[0]) if len(v) > 0 else 0.
                )
            ) for s, res in step.items()})
        return reversed(v)
```

If $|\mathcal{N}_t|$ is $O(m)$, then the running time of this algorithm is $O(m^2 \cdot T)$. However, note that it takes $O(m^2 \cdot k \cdot T)$ to convert the MDP to the π-implied MRP (where $|\mathcal{A}_t|$ is $O(k)$).

Now we move on to the Control problem - to calculate the Optimal Value Function and the Optimal Policy. Similar to the pattern seen so far, the Optimal Value Function

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

can be conveniently represented in terms of a sequence of Value Functions

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

for each of time steps $t = 0, 1, \ldots, T-1$, defined as:

$$V^*((t, s_t)) = V_t^*(s_t) \text{ for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t$$

Thus, the MDP State-Value Function Bellman Optimality Equation can be written as:

$$V_t^*(s_t) = \max_{a_t \in \mathcal{A}_t} \{\sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r_{t+1} \in \mathcal{D}_{t+1}} (\mathcal{P}_R)_t(s_t, a_t, r_{t+1}, s_{t+1}) \cdot (r_{t+1} + \gamma \cdot W_{t+1}^*(s_{t+1}))\} \quad \text{for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t \tag{1.9}$$

where

$$W_t^*(s_t) = \begin{cases} V_t^*(s_t) & \text{if } s_t \in \mathcal{N}_t \\ 0 & \text{if } s_t \in \mathcal{T}_t \end{cases} \text{ for all } t = 1, 2, \ldots, T$$

The associated Optimal (Deterministic) Policy

$$(\pi_D^*)_t : \mathcal{N}_t \to \mathcal{A}_t$$

is defined as:

$$(\pi_D^*)_t(s_t) = \arg\max_{a_t \in \mathcal{A}_t} \{\sum_{s_{t+1} \in \mathcal{S}_{t+1}} \sum_{r_{t+1} \in \mathcal{D}_{t+1}} (\mathcal{P}_R)_t(s_t, a_t, r_{t+1}, s_{t+1}) \cdot (r_{t+1} + \gamma \cdot W_{t+1}^*(s_{t+1}))\} \quad \text{for all } t = 0, 1, \ldots, T-1, s_t \in \mathcal{N}_t \tag{1.10}$$

So for a Finite MDP, this yields a simple algorithm to calculate V_t^* for all t, by simply decrementing down from $t = T-1$ to $t = 0$, using Equation (1.9) to calculate V_t^*, and Equation (1.10) to calculate $(\pi_D^*)_t$ for all $t = 0, 1, \ldots, T-1$ from the known values of W_{t+1}^* (since we are decrementing in time index t).

This algorithm is the adaptation of Value Iteration to the finite horizon case with this simple technique of "stepping back in time" (known as *Backward Induction*). Let's write some code to implement this algorithm. We are given a MDP over the augmented (finite) state space `WithTime[S]`. So this MDP is of type `FiniteMarkovDecisionProcess[WithTime[S], A]`. Our first task to to "unwrap" the state-reward probability transition function $\mathcal{P}_R$ of

this MDP into a time-indexed sequenced of state-reward probability transition functions $(\mathcal{P}_R)_t, t = 0, 1, \ldots, T-1$. This is accomplished by the following function `unwrap_finite_horizon_MDP` (`itertools.groupby` groups the augmented states by their time step, and the function `without_time` strips the time step from the augmented states when placing the states in $(\mathcal{P}_R)_t$, i.e., `Sequence[StateActionMapping[S, A]]`).

```
from itertools import groupby

ActionMapping = Mapping[A, StateReward[S]]
StateActionMapping = Mapping[NonTerminal[S], ActionMapping[A, S]]

def unwrap_finite_horizon_MDP(
    process: FiniteMarkovDecisionProcess[WithTime[S], A]
) -> Sequence[StateActionMapping[S, A]]:
    def time(x: WithTime[S]) -> int:
        return x.time

    def single_without_time(
        s_r: Tuple[State[WithTime[S]], float]
    ) -> Tuple[State[S], float]:
        if isinstance(s_r[0], NonTerminal):
            ret: Tuple[State[S], float] = (
                NonTerminal(s_r[0].state.state),
                s_r[1]
            )
        else:
            ret = (Terminal(s_r[0].state.state), s_r[1])
        return ret

    def without_time(arg: ActionMapping[A, WithTime[S]]) -> \
            ActionMapping[A, S]:
        return {a: sr_distr.map(single_without_time)
                for a, sr_distr in arg.items()}

    return [{NonTerminal(s.state): without_time(
        process.mapping[NonTerminal(s)]
    ) for s in states} for _, states in groupby(
        sorted(
            (nt.state for nt in process.non_terminal_states),
            key=time
        ),
        key=time
    )]
```

Now that we have the state-reward transition functions $(\mathcal{P}_R)_t$ arranged in the form of a `Sequence[StateActionMapping[S, A]]`, we are ready to perform backward induction to calculate V_t^*. The following function `optimal_vf_and_policy` accomplishes it with a straightforward use of Equations (1.9) and (1.10), as described above.

```
from operator import itemgetter

def optimal_vf_and_policy(
    steps: Sequence[StateActionMapping[S, A]],
    gamma: float
) -> Iterator[Tuple[V[S], FiniteDeterministicPolicy[S, A]]]:
    v_p: List[Tuple[V[S], FiniteDeterministicPolicy[S, A]]] = []

    for step in reversed(steps):
        this_v: Dict[NonTerminal[S], float] = {}
        this_a: Dict[S, A] = {}
        for s, actions_map in step.items():
            action_values = ((res.expectation(
                lambda s_r: s_r[1] + gamma * (
                    extended_vf(v_p[-1][0], s_r[0]) if len(v_p) > 0 else 0.
                )
            ), a) for a, res in actions_map.items())
```

```
            v_star, a_star = max(action_values, key=itemgetter(0))
            this_v[s] = v_star
            this_a[s.state] = a_star
        v_p.append((this_v, FiniteDeterministicPolicy(this_a)))
    return reversed(v_p)
```

If $|\mathcal{N}_t|$ is $O(m)$ for all t and $|\mathcal{A}_t|$ is $O(k)$, then the running time of this algorithm is $O(m^2 \cdot k \cdot T)$.

Note that these algorithms for finite-horizon finite MDPs do not require any "iterations to convergence" like we had for regular Policy Evaluation and Value Iteration. Rather, in these algorithms we simply walk back in time and immediately obtain the Value Function for each time step from the next time step's Value Function (which is already known since we walk back in time). This technique of "backpropagation of Value Function" goes by the name of *Backward Induction* algorithms, and is quite commonplace in many Financial applications (as we shall see later in this book). The above Backward Induction code is in the file rl/finite_horizon.py.

1.14 Dynamic Pricing for End-of-Life/End-of-Season of a Product

Now we consider a rather important business application - Dynamic Pricing. We consider the problem of Dynamic Pricing for the case of products that reach their end of life or at the end of a season after which we don't want to carry the product anymore. We need to adjust the prices up and down dynamically depending on how much inventory of the product you have, how many days remain for end-of-life/end-of-season, and your expectations of customer demand as a function of price adjustments. To make things concrete, assume you own a super-market and you are T days away from Halloween. You have just received M Halloween masks from your supplier and you won't be receiving any more inventory during these final T days. You want to dynamically set the selling price of the Halloween masks at the start of each day in a manner that maximizes your *Expected Total Sales Revenue* for Halloween masks from today until Halloween (assume no one will buy Halloween masks after Halloween).

Assume that for each of the T days, at the start of the day, you are required to select a price for that day from one of N prices $P_1, P_2, \ldots, P_N \in \mathbb{R}$, such that your selected price will be the selling price for all masks on that day. Assume that the customer demand for the number of Halloween masks on any day is governed by a Poisson probability distribution with mean $\lambda_i \in \mathbb{R}$ if you select that day's price to be P_i (where i is a choice among $1, 2, \ldots, N$). Note that on any given day, the demand could exceed the number of Halloween masks you have in the store, in which case the number of masks sold on that day will be equal to the number of masks you had at the start of that day.

A state for this MDP is given by a pair (t, I_t) where $t \in \{0, 1, \ldots, T\}$ denotes the time index and $I_t \in \{0, 1, \ldots, M\}$ denotes the inventory at time t. Using our notation from the previous section, $\mathcal{S}_t = \{0, 1, \ldots, M\}$ for all $t = 0, 1, \ldots, T$ so that $I_t \in \mathcal{S}_t$. $\mathcal{N}_t = \mathcal{S}_t$ for all $t = 0, 1, \ldots, T-1$ and $\mathcal{N}_T = \emptyset$. The action choices at time t can be represented by the choice of integers from 1 to N. Therefore, $\mathcal{A}_t = \{1, 2, \ldots, N\}$.

Note that:

$$I_0 = M, I_{t+1} = \max(0, I_t - d_t) \text{ for } 0 \leq t < T$$

where d_t is the random demand on day t governed by a Poisson distribution with mean λ_i if the action (index of the price choice) on day t is $i \in \mathcal{A}_t$. Also, note that the sales revenue on

day t is equal to $\min(I_t, d_t) \cdot P_i$. Therefore, the state-reward probability transition function for time index t

$$(\mathcal{P}_R)_t : \mathcal{N}_t \times \mathcal{A}_t \times \mathcal{D}_{t+1} \times \mathcal{S}_{t+1} \to [0, 1]$$

is defined as:

$$(\mathcal{P}_R)_t(I_t, i, r_{t+1}, I_t - k) = \begin{cases} \frac{e^{-\lambda_i}\lambda_i^k}{k!} & \text{if } k < I_t \text{ and } r_{t+1} = k \cdot P_i \\ \sum_{j=I_t}^{\infty} \frac{e^{-\lambda_i}\lambda_i^j}{j!} & \text{if } k = I_t \text{ and } r_{t+1} = k \cdot P_i \\ 0 & \text{otherwise} \end{cases}$$

for all $0 \leq t < T$

Using the definition of $(\mathcal{P}_R)_t$ and using the boundary condition $W_T^*(I_T) = 0$ for all $I_T \in \{0, 1, \ldots, M\}$, we can perform the backward induction algorithm to calculate V_t^* and associated optimal (deterministic) policy $(\pi_D^*)_t$ for all $0 \leq t < T$.

Now let's write some code to represent this Dynamic Programming problem as a `FiniteMarkovDecisionPro` and determine it's optimal policy, i.e., the Optimal (Dynamic) Price at time step t for any available level of inventory I_t. The type $\mathcal{N}_t$ is `int` and the type $\mathcal{A}_t$ is also `int`. So we create an MDP of type `FiniteMarkovDecisionProcess[WithTime[int], int]` (since the augmented state space is `WithTime[int]`). Our first task is to construct $\mathcal{P}_R$ of type:

```
Mapping[WithTime[int],
  Mapping[int, FiniteDistribution[Tuple[WithTime[int], float]]]]
```

In the class `ClearancePricingMDP` below, $\mathcal{P}_R$ is manufactured in `__init__` and is used to create the attribute `mdp: FiniteMarkovDecisionProces[WithTime[int], int]`. Since $\mathcal{P}_R$ is independent of time, we first create a single-step (time-invariant) MDP `single_step_mdp: FiniteMarkovDecisionProcess[int, int]` (think of this as the building-block MDP), and then use the function `finite_horizon_MDP` (from file rl/finite_horizon.py) to create `self.mdp` from `self.single_step_mdp`. The constructor argument `initial_inventory: int` represents the initial inventory M. The constructor argument `time_steps` represents the number of time steps T. The constructor argument `price_lambda_pairs` represents $[(P_i, \lambda_i) | 1 \leq i \leq N]$.

```
from scipy.stats import poisson
from rl.finite_horizon import finite_horizon_MDP

class ClearancePricingMDP:

    initial_inventory: int
    time_steps: int
    price_lambda_pairs: Sequence[Tuple[float, float]]
    single_step_mdp: FiniteMarkovDecisionProcess[int, int]
    mdp: FiniteMarkovDecisionProcess[WithTime[int], int]

    def __init__(
        self,
        initial_inventory: int,
        time_steps: int,
        price_lambda_pairs: Sequence[Tuple[float, float]]
    ):
        self.initial_inventory = initial_inventory
        self.time_steps = time_steps
        self.price_lambda_pairs = price_lambda_pairs
        distrs = [poisson(l) for _, l in price_lambda_pairs]
        prices = [p for p, _ in price_lambda_pairs]
        self.single_step_mdp: FiniteMarkovDecisionProcess[int, int] =\
            FiniteMarkovDecisionProcess({
                s: {i: Categorical(
```

```
                    {(s - k, prices[i] * k):
                     (distrs[i].pmf(k) if k < s else 1 - distrs[i].cdf(s - 1))
                     for k in range(s + 1)})
                    for i in range(len(prices))}
                for s in range(initial_inventory + 1)
            })
        self.mdp = finite_horizon_MDP(self.single_step_mdp, time_steps)
```

Now let's write two methods for this class:

- `get_vf_for_policy` that produces the Value Function for a given policy π, by first creating the π-implied MRP from `mdp`, then unwrapping the MRP into a sequence of state-reward transition probability functions $(\mathcal{P}_R^{\pi_t})_t$, and then performing backward induction using the previously-written function `evaluate` to calculate the Value Function.
- `get_optimal_vf_and_policy` that produces the Optimal Value Function and Optimal Policy, by first unwrapping `self.mdp` into a sequence of state-reward transition probability functions $(\mathcal{P}_R)_t$, and then performing backward induction using the previously-written function `optimal_vf_and_policy` to calculate the Optimal Value Function and Optimal Policy.

```
from rl.finite_horizon import evaluate, optimal_vf_and_policy
    def get_vf_for_policy(
        self,
        policy: FinitePolicy[WithTime[int], int]
    ) -> Iterator[V[int]]:
        mrp: FiniteMarkovRewardProcess[WithTime[int]] \
            = self.mdp.apply_finite_policy(policy)
        return evaluate(unwrap_finite_horizon_MRP(mrp), 1.)

    def get_optimal_vf_and_policy(self)\
            -> Iterator[Tuple[V[int], FiniteDeterministicPolicy[int, int]]]:
        return optimal_vf_and_policy(unwrap_finite_horizon_MDP(self.mdp), 1.)
```

Now let's create a simple instance of `ClearancePricingMDP` for $M = 12, T = 8$ and 4 price choices: "Full Price," "30% Off," "50% Off," "70% Off" with respective mean daily demand of $0.5, 1.0, 1.5, 2.5$.

```
ii = 12
steps = 8
pairs = [(1.0, 0.5), (0.7, 1.0), (0.5, 1.5), (0.3, 2.5)]
cp: ClearancePricingMDP = ClearancePricingMDP(
    initial_inventory=ii,
    time_steps=steps,
    price_lambda_pairs=pairs
)
```

Now let us calculate it's Value Function for a stationary policy that chooses "Full Price" if inventory is less than 2, otherwise "30% Off" if inventory is less than 5, otherwise "50% Off" if inventory is less than 8, otherwise "70% Off." Since we have a stationary policy, we can represent it as a single-step policy and combine it with the single-step MDP we had created above (attribute `single_step_mdp`) to create a `single_step_mrp: FiniteMarkovRewardProcess[int]`. Then we use the function `finite_horizon_mrp` (from file rl/finite_horizon.py) to create the entire (augmented state) MRP of type `FiniteMarkovRewardProcess[WithTime[int]]`. Finally, we unwrap this MRP into a sequence of state-reward transition probability functions and perform backward induction to calculate the Value Function for this stationary policy. Running the following code tells us that $V_0^\pi(12)$ is about 4.91 (assuming full price is 1),

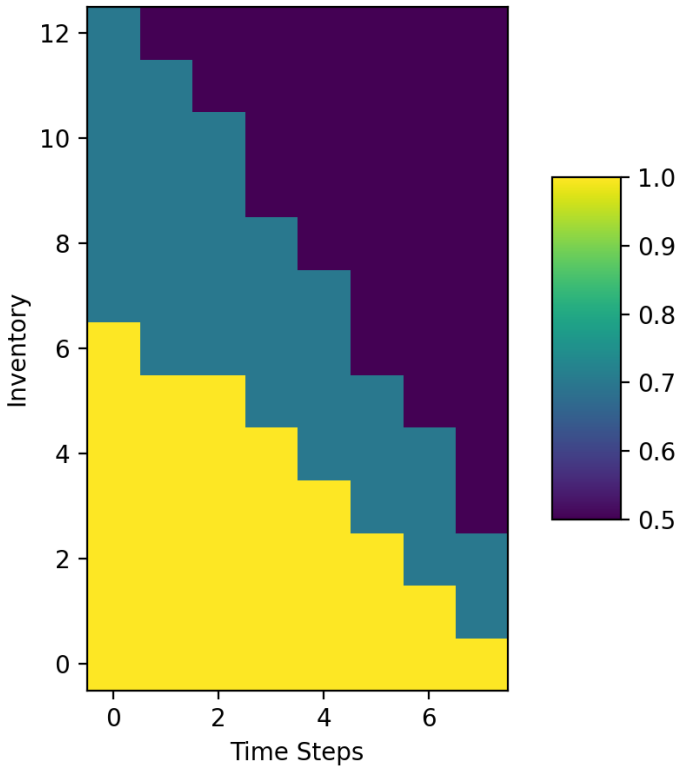

Figure 1.5: Optimal Policy Heatmap

which is the Expected Revenue one would obtain over 8 days, starting with an inventory of 12, and executing this stationary policy (under the assumed demand distributions as a function of the price choices).

```
def policy_func(x: int) -> int:
    return 0 if x < 2 else (1 if x < 5 else (2 if x < 8 else 3))

stationary_policy: FiniteDeterministicPolicy[int, int] = \
    FiniteDeterministicPolicy({s: policy_func(s) for s in range(ii + 1)})

single_step_mrp: FiniteMarkovRewardProcess[int] = \
    cp.single_step_mdp.apply_finite_policy(stationary_policy)

vf_for_policy: Iterator[V[int]] = evaluate(
    unwrap_finite_horizon_MRP(finite_horizon_MRP(single_step_mrp, steps)),
    1.
)
```

Now let us determine what is the Optimal Policy and Optimal Value Function for this instance of `ClearancePricingMDP`. Running `cp.get_optimal_vf_and_policy()` and evaluating the Optimal Value Function for time step 0 and inventory of 12, i.e. $V_0^*(12)$, gives us a value of 5.64, which is the Expected Revenue we'd obtain over the 8 days if we executed the Optimal Policy.

Now let us plot the Optimal Price as a function of time steps and inventory levels.

```
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
import numpy as np

prices = [[pairs[policy.act(s).value][0] for s in range(ii + 1)]
          for _, policy in cp.get_optimal_vf_and_policy()]

heatmap = plt.imshow(np.array(prices).T, origin='lower')
plt.colorbar(heatmap, shrink=0.5, aspect=5)
plt.xlabel("Time Steps")
plt.ylabel("Inventory")
plt.show()
```

Figure 1.5 shows us the image produced by the above code. The light shade is "Full Price," the medium shade is "30% Off" and the dark shade is "50% Off." This tells us that on day 0, the Optimal Price is "30% Off" (corresponding to State 12, i.e., for starting inventory $M = I_0 = 12$). However, if the starting inventory I_0 were less than 7, then the Optimal Price is "Full Price." This makes intuitive sense because the lower the inventory, the less inclination we'd have to cut prices. We see that the thresholds for price cuts shift as time progresses (as we move horizontally in the figure). For instance, on Day 5, we set "Full Price" only if inventory has dropped below 3 (this would happen if we had a good degree of sales on the first 5 days), we set "30% Off" if inventory is 3 or 4 or 5, and we set "50% Off" if inventory is greater than 5. So even if we sold 6 units in the first 5 days, we'd offer "50% Off" because we have only 3 days remaining now and 6 units of inventory left. This makes intuitive sense. We see that the thresholds shift even further as we move to Days 6 and 7. We encourage you to play with this simple application of Dynamic Pricing by changing $M, T, N, [(P_i, \lambda_i)|1 \leq i \leq N]$ and studying how the Optimal Value Function changes and more importantly, studying the thresholds of inventory (under optimality) for various choices of prices and how these thresholds vary as time progresses.

1.15 Generalizations to Non-Tabular Algorithms

The Finite MDP algorithms covered in this chapter are called "tabular" algorithms. The word "tabular" (for "table") refers to the fact that the MDP is specified in the form of a finite data structure and the Value Function is also represented as a finite "table" of non-terminal states and values. These tabular algorithms typically make a sweep through all non-terminal states in each iteration to update the Value Function. This is not possible for large state spaces or infinite state spaces where we need some function approximation for the Value Function. The good news is that we can modify each of these tabular algorithms such that instead of sweeping through all the non-terminal states at each step, we simply sample an appropriate subset of non-terminal states, calculate the values for these sampled states with the appropriate Bellman calculations (just like in the tabular algorithms), and then create/update a function approximation (for the Value Function) with the sampled states' calculated values. The important point is that the fundamental structure of the algorithms and the fundamental principles (Fixed-Point and Bellman Operators) are still the same when we generalize from these tabular algorithms to function approximation-based algorithms. In Chapter **??**, we cover generalizations of these Dynamic Programming algorithms from tabular methods to function approximation methods. We call these algorithms *Approximate Dynamic Programming*.

We finish this chapter by refering you to the various excellent papers and books by Dimitri Bertsekas - (Dimitri P. Bertsekas 1981), (Dimitri P. Bertsekas 1983), (Dimitri P. Bertsekas 2005), (Dimitri P. Bertsekas 2012), (D. P. Bertsekas and Tsitsiklis 1996) - for a comprehensive treatment of the variants of DP, including Asynchronous DP, Finite-Horizon DP and Approximate DP.

1.16 Summary of Key Learnings from this Chapter

Before we end this chapter, we'd like to highlight the three highly important concepts we learnt in this chapter:

- Fixed-Point of Functions and Banach Fixed-Point Theorem: The simple concept of Fixed-Point of Functions that is profound in its applications, and the Banach Fixed-Point Theorem that enables us to construct iterative algorithms to solve problems with fixed-point formulations.
- Generalized Policy Iteration: The powerful idea of alternating between *any* method for Policy Evaluation and *any* method for Policy Improvement, including methods that are partial applications of Policy Evaluation or Policy Improvement. This generalized perspective unifies almost all of the algorithms that solve MDP Control problems.
- Backward Induction: A straightforward method to solve finite-horizon MDPs by simply backpropagating the Value Function from the horizon-end to the start.

Bibliography

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.

Bertsekas, D. P., and J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific.

Bertsekas, Dimitri P. 1981. "Distributed Dynamic Programming." In *1981 20th IEEE Conference on Decision and Control Including the Symposium on Adaptive Processes*, 774–79. https://doi.org/10.1109/CDC.1981.269319.

———. 1983. "Distributed Asynchronous Computation of Fixed Points." *Mathematical Programming* 27: 107–20.

———. 2005. *Dynamic Programming and Optimal Control, Volume 1, 3rd Edition.* Athena Scientific.

———. 2012. *Dynamic Programming and Optimal Control, Volume 2: Approximate Dynamic Programming*. Athena Scientific.

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

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.