From Markov Decision Processes to Reinforcement Learning with Python

Saúl Díaz Infante, David González Sánchez

2024-05-30

Table of contents

Preface

This notes are based in the course from Berstekas for the MIT see all lectures and other resources for complete the understanding.

Outline

The textbook for chapter one is Bertsekas' book (Bertsekas 2005). Chapters 2 and 3 are adapted from Sutton's book (Ch. 3, Ch. 4, Sutton and Barto 2018). For application and broad connection with more machine learning applications, we refer to (Brunton and Kutz 2019). Also, we recommend a handbook of algorithms (Szepesvári 2022). For applications with implemented code, we follow the books (Bilgin 2020).

1 The Dynamic Programming Algorithm

1.1 Introduction

1.2 The Basic Problem

1.3 The Dynamic Programming Algorithm

1.4 State Augmentation and Other Reformulations

1.5 Some Mathematical Issues

1.6 Dynamic Programming and Minimax Control

1.7 Notes, Sources, and Exercises

2 Finite Markov Decision Processes (MDPs)

That is what ChatGPT would answer to a 5-year-old kid. Alright, let's imagine you have a little robot friend named Robo. Robo likes to explore and do different things, but Robo doesn't always know what to do next. A Markov Decision Process (MDP) is like giving Robo a set of rules to help it decide what to do next based on where it is and what it knows.

Imagine Robo is in a room full of toys. Each toy is like a different choice Robo can make, like playing with blocks or reading a book. But Robo can't see the whole room at once, so it has to decide what to do based on what it can see and remember.

In an MDP, Robo learns from its past experiences. If it finds that playing with blocks usually makes it happy, it's more likely to choose that again next time. But if it tries reading a book and doesn't like it, it might choose something else next time.

So, a Markov Decision Process helps Robo make decisions by learning from what it's done before and what it can see around it, kind of like how you learn from playing with different toys and remembering which ones you like best.

2.1 The Agent–Environment Interface

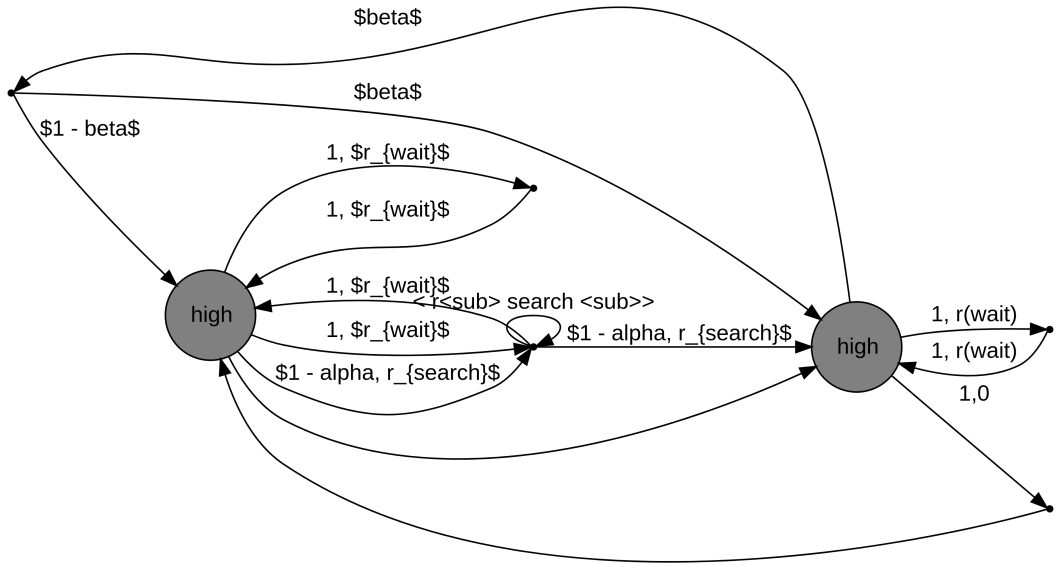
Example 2.1. A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and an arm and gripper that can pick them up and place them in an onboard bin; it runs on a rechargeable battery. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent based on the current charge level of the battery. To make a simple example, we assume that only two charge levels can be distinguished, comprising a small state set $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. In each state, the agent can decide whether to

1. actively **search** for a can for a certain period of time,
2. remain stationary and **wait** for someone to bring it a can, or
3. head back to its home base to **recharge** its battery.

When the energy level is **high**, recharging would always be foolis h, so we do not include it in the action set for this state. The action sets are then $\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$ and $\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$.

The rewards are zero most of the time, but become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. The best way to find cans is to actively search for them, but this runs down the robot's battery, whereas waiting does not. Whenever the robot is searching, the possibility exists that its battery will become depleted. In this case the robot must shut down and wait to be rescued (producing a low reward).

If the energy level is **high**, then a period of active search can always be completed without risk of depleting the battery. A period of searching that begins with a **high** energy level leaves the energy level high **with** probability α and reduces it to low with probability $1 - \alpha$. On the other hand, a period of searching undertaken when the energy level is low leaves it low with probability β and depletes the battery with probability $1 - \beta$. In the latter case, the robot must be rescued, and the battery is then recharged back to **high**. Each can collected by the robot counts as a unit reward, whereas a reward of -3 results whenever the robot has to be rescued. Let $r_{\texttt{search}}$ and $r_{\texttt{wait}}$, with $r_{\texttt{search}} > r_{\texttt{wait}}$, denote the expected numbers of cans the robot will collect (and hence the expected reward) while searching and while waiting respectively. Finally, suppose that no cans can be collected during a run home for recharging, and that no cans can be collected on a step in which the battery is depleted. This system is then a finite MDP, and we can write down the transition probabilities and the expected rewards, with dynamics as indicated in the table on the left:



Exercise 2.1. Give a table analogous to that in (p.53, Ex. 3.3, Sutton and Barto 2018), but for $p(s_0, r|s, a)$. It should have columns for s, a, s_0 , r, and $p(s_0, r|s, a)$, and a row for every 4-tuple for which $p(s_0, r|s, a) > 0$.

2.2 Goals and Rewards

2.3 Returns and Episodes

2.4 Unified Notation for Episodic and Continuing Tasks

2.5 Policies and Value Functions

2.6 Optimal Policies and Optimal Value Functions

2.7 Optimality and Approximation

2.8 Summary

3 Dynamic Programming

An explanation from ChatGPT. Alright, imagine you have a big puzzle to solve, but it's too big for you to finish in one go. So, you decide to break it into smaller puzzles, and you solve each of these small puzzles one by one. But, here's the clever part: as you solve these small puzzles, you remember the solutions. That way, if you come across the same small puzzle again, you don't have to solve it all over again. You already know the answer! Dynamic programming is like solving a big puzzle by breaking it into smaller ones and remembering the solutions to the smaller ones to make solving the big puzzle easier and faster.

3.1 Policy Evaluation (Prediction)

3.2 Policy Improvement

3.3 Policy Iteration

3.4 Value Iteration

3.5 Asynchronous Dynamic Programming

3.6 Generalized Policy Iteration

3.7 Efficiency of Dynamic Programming

3.8 Summary

4 Applications

4.1 Recycling Robot

4.2 A robot with randomly moves in a grid world.

References

Bertsekas, Dimitri P. 2005. *Dynamic Programming and Optimal Control. Vol. I.* Third. Athena Scientific, Belmont, MA.

Bilgin, E. 2020. *Mastering Reinforcement Learning with Python: Build Next-Generation, Self-Learning Models Using Reinforcement Learning Techniques and Best Practices*. Packt Publishing. https://books.google.com.mx/books?id=s0MQEAAAQBAJ.

Brunton, Steven L., and J. Nathan Kutz. 2019. *Data-Driven Science and Engineering*. Cambridge University Press, Cambridge. https://doi.org/10.1017/9781108380690.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA.

Szepesvári, Csaba. 2022. *Algorithms for Reinforcement Learning*. Vol. 9. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer, Cham. https://doi.org/10.1007/978-3-031-01551-9.