COMP 138 RL: Programming Assignment 2

Jane Slagle

October 21, 2024

1 Introduction

1.1 Problem Background

This programming assignment is centered around Monte Carlo (MC) control methods, specifically MC control methods for computing the optimal policy. Unlike other reinforcement learning (RL) methods such as Markov Decision Processes, which assume full knowledge of the environment, MC methods require only experience (in the form of sequences of states, actions, and rewards) from actual or simulated interactions with the environment. Despite not having prior knowledge of the environment's transition probabilities or reward structure, MC methods can still achieve optimal behavior by learning from the actual sampled experience.

MC methods are generally divided into two main categories based on their objectives, MC methods for prediction and MC methods for control. MC methods for prediction focus on evaluating how good a policy is by estimating the value function for that policy. MC methods for control on the other hand aim to improve the policy, serving the purpose of approximating the optimal policy in the learning process. In general, MC methods for control start with an initial policy and Q-function (which estimates the expected sum of discounted rewards for taking action a in state s and then following the policy afterward). The return is computed for each state, action (s, a) pair and the Q-function is updated based on those returns. With this process, the policy is improved, being updated each episode, ultimately leading to the optimal policy.

1.2 Relation of Problem to Reinforcement Learning

MC methods solve reinforcement learning problems by averaging the sampled returns for each state, action pair. The main difference between MC methods and other RL approaches that also involve averaging returns, such as bandit problems which average rewards for each action, is that MC methods account for multiple states.

MC methods are used for episodic tasks, where each episode lasts for a given amount of time steps. Value estimate and policies are updated only after the completion of an entire episode, rather than after each time step which is the

case with other RL methods, such as Temporal Difference (TD) methods. Thus, MC methods are incremental in the sense that learning occurs on an episode by episode basis.

2 Problem

2.1 Problem Statement

Consider driving a race car around a turn like those shown in Figure 1 (see below), taken from the book [1]. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by +1, -1, or 0 in each step, for a total of nine (3 x 3) actions. Both velocity components are restricted to be non negative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the episode continues. Before updating the car's location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode ends; if it intersects anywhere else, the car is considered to have hit the track boundary and is sent back to the starting line. To make the task more challenging, with probability 0.1 at each time step the velocity increments are both zero, independently of the intended increments. Apply a Monte Carlo control method to this task to compute the optimal policy for each starting state. Exhibit several trajectories following the optimal policy (but turn the noise off for those trajectories).

Figure 1: A couple of right turns for the racetrack task (this is figure 5.5 in the book [1]).

2.2 Approach

In this assignment, we will implement the two racetracks given above in Figure 1, simulating a car driving from each of the starting line positions to the finish line on the two tracks when following an optimal policy that we will have found in the simulation process. The agent, which represents a car driving through the racetrack, will learn how best to navigate through each track, using trial and error from each possible start line position with the goal of finding the optimal policy that will allow it to reach the finish line as quickly as possible while staying within the boundary walls of the racetrack.

With our approach, the racetrack is represented as a grid where each grid cell represents one of the possible positions of the car on the racetrack - a wall that the car cannot drive through, a valid part of the track that the car can drive through, the starting line positions or the finish line which the car has the goal of hitting while driving through the track. The car's state is defined by both its position, (x, y), on the racetrack and its velocity, (vx, vy). The velocity represents how the car moves in each possible direction at each time step on it's path through the racetrack. The car controls its velocity by adjusting its horizontal and vertical components with three different possible actions - $+1$ (increase the speed), -1 (decrease the speed), or 0 (maintain the current speed, no movement taken). This gives a total of 9 different possible actions for each velocity component at each time step. The velocity must be between 0 and 4 which means that the car cannot move backwards on its path through the racetrack and it also cannot drive too fast.

Each episode starts with the car at a random position on the start line with the car's velocity set to be zero in both components. The car moves according

to its velocity and takes actions that adjust its speed to allow it to drive through the track. Each episode starts when the car takes off from the start line and only ends when the car crosses the finish line. While driving in each episode, the car continuously checks for intersections with either the boundary walls of the track or the finish line and acts accordingly. The car explores different actions (movements) and learns from the results of each action based on the feedback from the environment, given in the form of a reward. A reward of -1 signals that the car did not cross the finish line, the negative reward given to incentives finishing as fast as possible. A reward of 0 is given to represent that the car did cross the finish line, it's non-negative value indicating that the car is doing something that we want it to.

The objective of the task here is to use an MC control method to learn the optimal policy for the car. This means that we wish to find a policy that will allow the car to reach the finish line in the least amount of time without hitting any boundary walls on the racetrack. MC control methods involve learning an optimal policy through trial and error by running episodes, gathering experience (the state, action, and reward values) and updating the policy based on the experience of that entire episode. So, in our approach, over multiple episodes, the car learns the best actions to take from any starting position on the racetrack to reach the finish line. This learning is done using off-policy MC control with an epsilon-greedy behavior policy. More specifically, we will follow the approach outlined in the following algorithm given in Section 5.7 (page 111) of the book [1].

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Figure 2: Off-policy MC Control Algorithm, given in Section 5.7 (page 111) of the book [1]). This is the MC control method that our agent will follow in our approach, see Section on Agent.py below.

2.2.1 Off-Policy MC Control

As stated above, we use an off-policy MC control approach for this task, specifically, following the algorithm outlined above in Figure 2. Off-policy methods have both a behavior and target policy. In the above algorithm, b represents the behavior policy while π represents the target policy. The behavior policy is used to generate the behavior observed in the experiences (which are the sequences of states, actions, and rewards from each time step in the episode). The target policy is the one used for policy evaluation and improvement, it is the optimal policy.

In Section 5.7 of the book, it is stated that the behavior policy should be a soft policy. Thus, in our approach, we have chosen the behavior policy, b, to be an epsilon-greedy policy. This means that the agent, which is the car here, will generate episodes using b and with probability epsilon will choose an action randomly, and with probability $1 - epsilon$ will choose a greedy action, which means it will choose the best action to take.

Since we are using an MC control approach, in the code described below in Section 3, we are gathering experiences (the sequences of states, actions and rewards) throughout each entire episode, starting from the initial start state on the start line until the agent (the car) reaches the finish line. After each episode is completed, the agent then calculates the total reward from each state, action pair that it visited during the episode. The total reward is then used to update the Q values for that state, action pair.

With the above algorithm, our MC control method is based on Generalized Policy Improvement (GPI) and uses weighted importance sampling to estimate the optimal policy and optimal Q function, action-value function, values:

GPI Our optimal policy is updated using a GPI approach. GPI says that if you have an estimate of the policy value (the Q function value), then the policy can be improved by acting greedily with respect to that Q function value. This is exactly what we observe in the algorithm given in Figure 2 when we look at how $pi(S_t)$ is updated in each episode. More specifically, with this part of the algorithm:

$$\pi(S_t) \leftarrow argmax_a Q(S_t, a)$$

Weighted Importance Sampling Weighted importance sampling was used in our approach to help in updating the estimates based on the experiences generated from b. We see the weights, W, being used in the above algorithm to update C and Q. The weight used here is given by the following:

$$W = W \frac{1}{b(A_t|S_t)}$$

where the weight W adjusts the contribution of each episode's return based on how likely b is to have taken action A_t in the state.

2.3 Motivation Behind Approach

Overall, the agent in our implementation follows an off-policy MC control algorithm with an epsilon-greedy behavior policy to learn the optimal policy for reaching the finish line from any possible start position on the start line.

2.3.1 Why use an off-policy MC control algorithm approach?

The main motivation behind using an off-policy MC control algorithm in our approach with the agent comes from how the algorithm in Figure 2 was given in Section 5.7 of the book which is also the section where this problem was introduced, so I figured that this algorithm would be a fitting approach to use for the problem.

In addition, with this approach, there is a separation of policies into behavior and target policies. This is a wanted approach as separating the policies like so allow us to have a behavior policy that gains experience from all possible actions, and then we can create the target policy, which is the one that we actually use in our results, that is greedy.

MC control methods also mean that updates are only made after an entire episode has finished. This means that our agent, the car, learns the actions based on complete trajectory paths (the entire path of the car from start to finish through the racetrack) without needing to know the entire dynamics of the racetrack beforehand. Since the policy is being updated based on the results

of many episodes, our approach is able to refine its strategy over time which allows it to minimize mistakes, such as hitting boundary walls on the track, which will enable it to reach the finish line faster.

2.3.2 Why choose b to be epsilon-greedy?

Choosing a behavior policy b that is epsilon-greedy allows the agent to balance exploration and exploitation where exploration is trying new actions and exploitation is choosing the action that is known to be the best. By finding a balance between the two, our agent is able to still see opportunities for better strategies. By usually following the action that will give the best results, but also having the option to learn more about the environment, our agent will be better able to perform overall.

2.3.3 Why use GPI and weighted importance sampling?

We use GPI in how we both evaluate and improve the policy. We first evaluate the policy using the Q function value and then we improve the policy by acting greedily based on the current Q function values. We are iteratively evaluating and improving the policy, which means that with GPI, we are converging to the optimal policy gradually as the agent interacts more with the environment.

By using weighted importance sampling, we are adjusting the rewards based on the behavior policy so that the rewards more accurately represent what would happen with the target policy. In the formula for W, A_t are actions that are found in the target policy, pi. This means that we are weighting the rewards based on the likelihood of actions that are taken by the target policy, which is the optimal policy here. This helps the agent be able to learn better in the environment.

3 Code

I have structured my approach to this problem into three different files: *Racetrack.py*, *Agent.py* and *main.py*.

Racetrack.py sets up the environment that the agent (which is the car here) interacts with. The agent then learns how best to drive through the racetrack environment in *Agent.py*. The racetrack and agent are brought together in *main.py* which simulates the agent (car) driving through the environment (racetrack) and visualizes the trajectories of the agent from each of the different starting positions when following the learned optimal policy.

3.1 Racetrack.py

This class is responsible for building the racetrack environment. It does so by defining the layout of the racetrack, controlling the movement of a car on the racetrack based on the actions that the car is able to take, detects possible collisions of the car on the racetrack with walls, keeps track of whether the

car crosses the finish line and also computes the trajectories of the car from each starting position on the racetrack based on the optimal policy found in the Agent class.

The racetrack layout is inputted as a list of strings, and the class then builds a numerical representation of the racetrack that it is then able to interact with. The class can build any racetrack is inputted with the correct representation of strings. The purpose of this class is to provide a representation of the racetrack and to also define what happens when an agent (car) takes an action at any position on the racetrack, including how the velocity of the car would change, position of the car would update and how to handle how the car actually moves (hitting boundary walls, crossing the finish line or driving on a valid part of the racetrack).

3.2 Agent.py

This class is responsible for setting up the agent that interacts with the racetrack environment. The agent here represents a car driving through the racetrack. The racetrack is given as input and the agent then creates the racetrack environment that it will interact with using the Racetrack class. The purpose of the Agent class is to guide the car through the racetrack environment while also learning the optimal policy to take based on the state, action values. This class directly follows the off-policy MC control for estimating optimal policy algorithm given in Figure 2.

3.3 main.py

In this file, we run a simulation over N episodes (in this case, N = 300,000) of a car driving through a racetrack with the optimal policy learned from the agent. This file is responsible for initializing a racetrack, creating an agent which is the car and then running a simulation of the car through the racetrack. In the simulation, the car loops over all episodes through the racetrack and learns the optimal policy. The optimal policy is stored in the *agent.pi* attribute of the Agent class, so once that attribute is found after running through all episodes, *main.py* then uses it to create find the trajectories with that optimal policy and visualize them for each starting position on the racetrack.

I ran the simulation with 300,000 episodes so that the agent would be able to learn from a wide range of experiences, which would hopefully lead to it being able to better learn a more optimal policy.

4 Results

We visualized the results from this task by plotting the trajectory of the car from each possible starting position on the two racetracks provided in Figure 1. The trajectories that we used to create the following plots, calculated in *Racetrack.py* are dictionaries that store the trajectory for all starting positions

on the start line for the given racetrack that we are plotting. Each trajectory is a list of tuples where the first element in the tuple is the car's position at that time step and the second is the action that the car took at that position to move to the next state. Thus, along with the visualization of each starting position's trajectory, we will also use the actions taken at each point to gain some insight into how the car was driving (if it slowed down going around corners, slowed down when approaching the finish line to avoid hitting walls, etc.).

4.1 Racetrack One Results

4.1.1 Visualization Plot Results

Figure 3 below shows the trajectory from each of the 6 different possible starting states on the starting line of the below racetrack. It shows the trajectory that the car takes from each starting position to reach the finish line using the optimal policy.

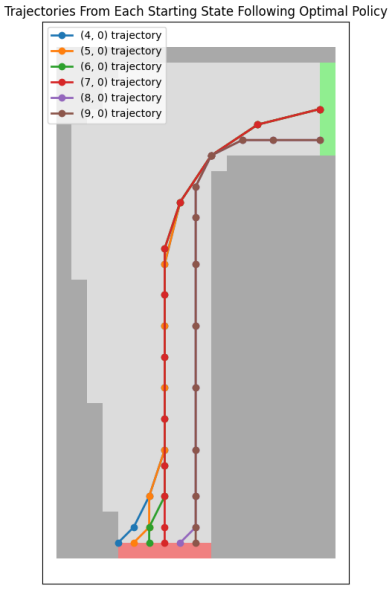

Figure 3: Trajectories from each starting state following the optimal policy in the racetrack given on the left side of Figure 1.

Based on the plot, I believe that the car is taking an optimal path from each of the six different points on the starting line. For instance, for the points on the starting line that start the farthest away from the finish line, such as (4,0), (5,0) and (6,0), they all move over to the right pretty early on in the racetrack, basically immediately, because doing so will bring them to a position right away that is already closer to the finish line. This indicates that the car is optimizing its path to the finish line in each of those.

I also believe these paths to be optimal as each of them pass closely to the corner in the racetrack, as close to it as they can without hitting a boundary wall on the track. Passing so closely to the corner like that allows the car in each trajectory to reach the finish line sooner because it allows the car to take a shorter path from the start line to the finish line.

4.1.2 Analysis of Actions Corresponding with Figure 3 Results

We will now take a look at the actions that the car took at each trajectory, and view the actions with Figure 3 to gain some insight into how the car was moving on each path.

There are six different starting positions on the start line for racetrack one, so we will choose only one starting position point out of those six to analyze and gain some insight from. We will analyze the actions from start position $(4, 0)$.

From running the simulation with racetrack one in *main.py*, we got the results of all trajectories from each starting position. Using those results, we are able to create the following table for start position $(4, 0)$ where each *step* corresponds to each point plotted in Figure 3 and the *position* represents where the car is moving to at each step while *action* represents what action the car took at each step to make it to the next step.

Table 1: Steps and Actions for Starting Position (4, 0)

Step	**Position (x, y)**	**Action (ax, ay)**
1	(4, 0)	(1, 1)
2	(5, 1)	(0, 1)
3	(6, 3)	(0, 1)
4	(7, 6)	(-1, 1)
5	(7, 10)	(-1, 1)
6	(7, 14)	(-1, 0)
7	(7, 18)	(1, 1)
8	(8, 22)	(1, -1)
9	(10, 25)	(1, -1)
10	(13, 27)	(1, -1)
11	(17, 28)	None

Looking at the above actions in Table 1 and looking at them in accordance with the car's path from $(4, 0)$ in Figure 3, we observe that as the car moves right from it's original starting position on the racetrack, that it slows down in

the x direction (as indicated by -1 in the actions). This is due to how when the car is moving right, it is moving closer the boundary walls of the track that are on the right side over there. So, the car is slowing down so that it does not hit the wall. In addition, as the car rounds the corner (which is the actions that it takes closer to the end of it's trajectory), we note that it slows down in the y direction. This is probably so that it is able to round the corner without overshooting and hitting any walls or going out of bounds.

4.2 Racetrack Two Results

Figure 4 below shows the trajectory from each of the 23 different possible starting states on the starting line of the below racetrack. It shows the trajectory that the car takes from each starting position to reach the finish line using the optimal policy.

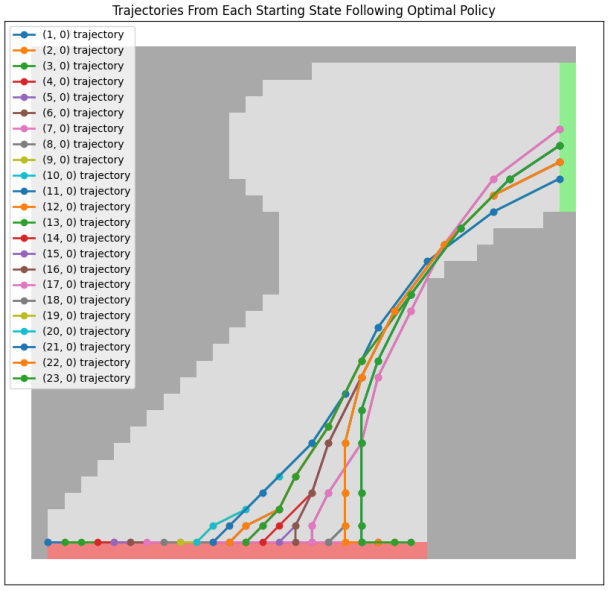

Figure 4: Trajectories from each starting state following the optimal policy in the racetrack given on the right side of Figure 1.

Based on the plot, I believe that the car is taking an optimal path from each of the 23 different points on the starting line. For instance, for the points that start the farthest away from the finish line, such as $(1, 0) - (10, 0)$, they all immediately move over to the right as their first movements, moving right so that they can get the closest that they are able to do so right away to the finish line. Moving right at such an early stage allows the car at each of those starting points to be a lot closer to the finish line. This indicates that the car is optimizing its path to the finish line in each of those starting positions. In addition, the same can be said for the last few starting points, $(20, 0) - (23, 0)$ who move similarly, but this time to the left so that when they move to the finish line, they are not going to move straight up and hit the boundary walls of the track.

Like in Figure 3, the trajectories plotted here in Figure 4 also move as close to the corners as they can without hitting the track boundary wall. The car moves so close to the corners because that is the shortest path it is able to take from the start to the finish.

5 Conclusion

In conclusion, the patterns identified from the results given in Figures 3 and 4 indicate that the car is taking optimal trajectory paths from each possible starting position to the finish line for each racetrack. I do think that the results could be more streamlined, I think the trajectories could be even more optimized. To achieve more optimized results, I would need to run the simulation with more episodes. Running more episodes would help the agent (the car) be able to learn more from the experiences which could help lead to an even more optimized policy. However, by introducing more episodes, then I would also be introducing more computational cost, so there is a trade-off involved there.

That being said though, I do think that Figures 3 and 4 and Table 1 support that I have found an optimization strategy that a car can follow for reaching the finish line in a timely and fast manner for any inputted racetrack. As touched upon in Section 4, there were consistent patterns observed for the results from both racetracks where the points that were farthest away from the finish line would immediately make movements that would bring them closer to the finish line and optimize their paths. This indicates that the agent used in my approach has an underlying strategic approach to minimize the path it is taking through the racetrack. These initial movements are so indicative because they clearly show how the car is positioning itself closer to the most direct path leading to the finish line, supporting that it is following an efficient trajectory.

We observe careful movements around the racetrack boundaries in Figures 3 and 4, highlighting how the agent is able to optimize avoiding collisions with the track boundaries while also reducing the overall distance of it's travelled path. I think that the car's ability to stay close to the corners of the track without actually hitting them shows it's trajectory optimization.

Overall, my results seem to be on the right track - pun intended!

6 Extra Credit

For the extra credit portion of this assignment, I completed exercises 5.6 and 6.8 from Chapters 5 and 6 of Sutton and Barto. Both are non-programming tasks that were not covered in the live Zoom meetings nor the pre-recorded lectures.

6.1 Exercise 5.6

Exercises 5.6 states:

> What is the equation analogous to (5.6) for action values $Q(s, a)$ instead of state values $V(s)$, again given returns generated using b?

Solution:
We want state, action value pairs (s, a) instead of just state values s, so equation 5.6 becomes:

$$Q(s,a) = \frac{\sum_{t \in T(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t \in T(s,a)} \rho_{t:T(t)-1}}$$

6.2 Exercise 6.8

Exercises 6.8 states:

> Show that an action-value version of (6.6) holds for the action-value form of the TD $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$ again assuming that values don't change from step to step.

Solution:
Equation (6.6) is given in Section 6.1 from the book [1] as:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &\dots \\ &= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \end{aligned}$$

We want to use $Q_(S_t, A_t)$ instead of $V(S_t)$. This gives us:

$$\begin{aligned} G_t - Q(S_t, A_t) &= R_{t+1} + \gamma G_{t+1} - Q(S_t, A_t) + \gamma Q(S_{t+1}, A_{t+1}) - \gamma Q(S_{t+1}, A_{t+1}) \\ &= \delta_t + \gamma(G_{t+1} - Q(S_{t+1}, A_{t+1})) \end{aligned}$$

where $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$
By the same logic we can repeat the same process for $G_{t+1} - Q(S_{t+1}, A_{t+1})$:

$$G_t - Q(S_{t+1}, A_{t+1}) = \delta_{t+1} + \gamma G_{t+2} - Q(S_{t+2}, A_{t+2})$$

so substituting this back into the original equation we get:

$$G_t - Q(S_t, A_t) = \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - Q(S_{t+2}, A_{t+2}))$$

If we follow this same pattern of recursively expanding further and further we will eventually get:

$$G_t - Q(S_t, A_t) = \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k$$

Thus, we have shown that the action-value version of equation 6.6 is also equal to the sum of discounted TD errors δ_k.

References

[1] Richard S. Sutton and Barto Andrew G. *Reinforcement learning : an introduction*. The MIT Press, 2018.