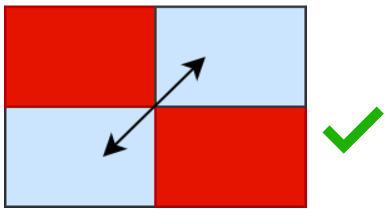
Project Report

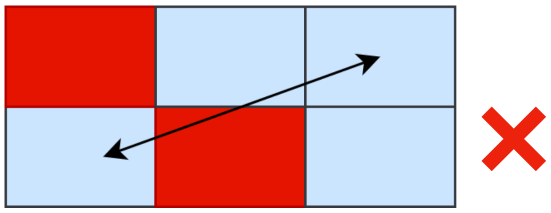
Keshav Goyal

Algorithms Used and Assumptions:

Assumptions:

- I have assumed that the car travels along its velocity vector.
- If the vector passes through a wall then its a wall hit.
- And that being tangent to the corner of a wall is fine.
- So, the following is followed:





- Car keeps moving with the same velocity as current state, during this travel time the action is applied and when it reaches the new state the velocity is changed.

Basic Code Structure:

- There is a class called "Environment" which contains information about the track and given a state and action gives back the next state and reward to the agent.
- There are 2 helper functions, the act_map() function gives the action value given the action's index and state_map() converts a state from list to string format.
- There are 2 separate classes for On policy and Off Policy agents.
- There is a function "visualise()" which uses pygame module to create a visualisation of the given trajectory.

On-Policy Monte Carlo:

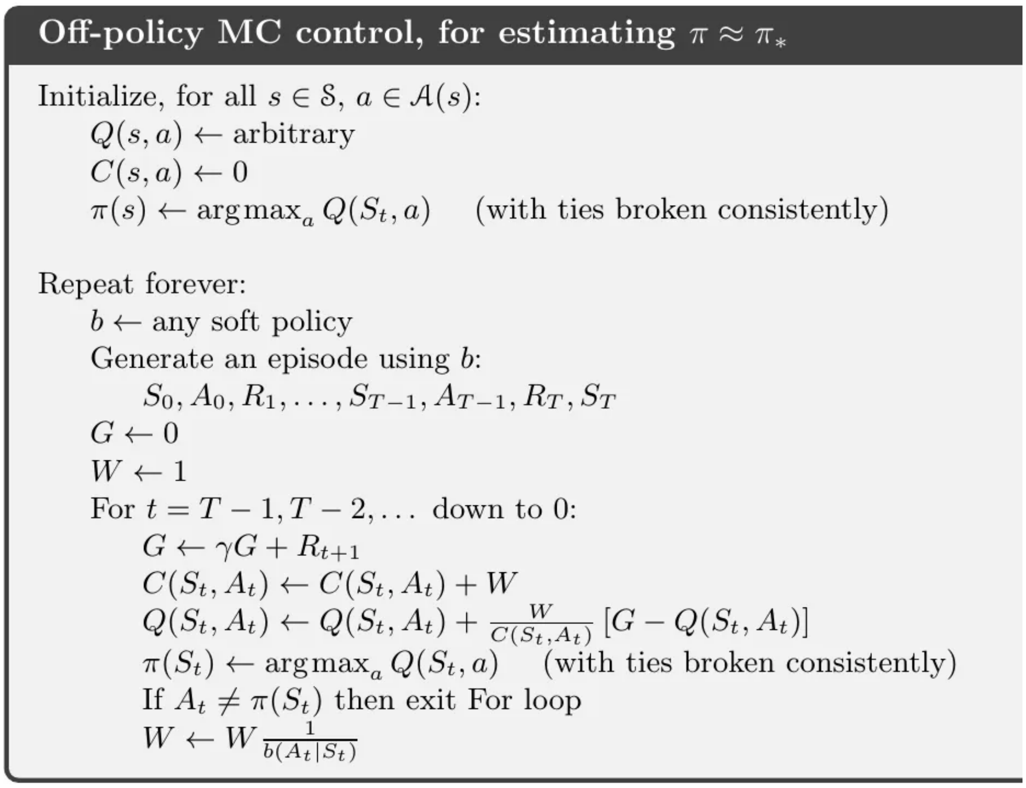
- I have used the ϵ-soft first-visit On-Policy Monte Carlo Algorithm
- The basic algorithm skeleton is:

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list
    $\pi(a|s) \leftarrow$ an arbitrary $\varepsilon$-soft policy

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each pair $s,a$ appearing in the episode:
            $G \leftarrow$ return following the first occurrence of $s,a$
            Append $G$ to $Returns(s,a)$
            $Q(s,a) \leftarrow \text{average}(Returns(s,a))$
    (c) For each $s$ in the episode:
            $a^* \leftarrow \arg\max_a Q(s,a)$
            For all $a \in \mathcal{A}(s)$:
                $\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
```

- Although, some modifications have been made to get better convergence:
  - The ϵ-soft policy has been modified such that if there is an action that has not been experienced for given state, then that action is chosen. Otherwise normal soft approach is followed.
  - There is a -10 reward for hitting a wall.
  - Gamma = 0.9
  - Epsilon = 0.6 for first 50 episodes and 0.3 for rest.
  - Total 120 episodes are run.
  - Epsilon is high for first few episodes to increase exploration.
  - Another approach I tried for more exploration is to start with epsilon = 1 and gradually decrease which gave similar results as the chosen approach.
  - An approach for faster convergence was to reduce exploration for initial steps of an episode after some episodes, in order to explore more in future steps while keeping episode length small. This approach did not work as expected but I suspect the issue was a bug in my return calculation and not the idea itself.

Off-Policy Monte Carlo:

- I have used the every-visit Off-Policy weighted importance sampling Monte Carlo Algorithm.
- The basic skeleton is:



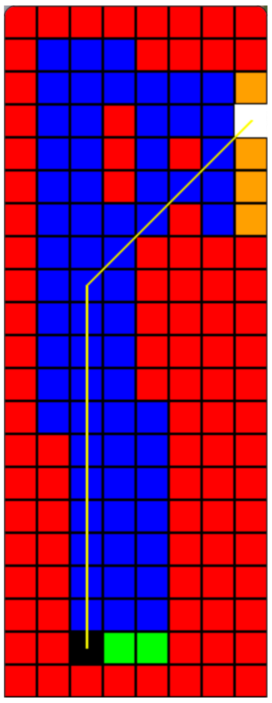
```
Off-policy MC control, for estimating $\pi \approx \pi_*$

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

Repeat forever:
    $b \leftarrow$ any soft policy
    Generate an episode using $b$:
        $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    For $t = T-1, T-2, \ldots$ down to 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 For loop
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

- This algorithm makes the behaviour policy into an epsilon-soft policy of the target policy. This is done to keep the behaviour policy related to the target policy for faster convergence.
- Parameters:
  - Not many modifications were made to the algorithm.
  - There is a -10 reward for hitting a wall and Gamma = 0.9
  - Epsilon = 0.3 for all the 50,000 episodes.

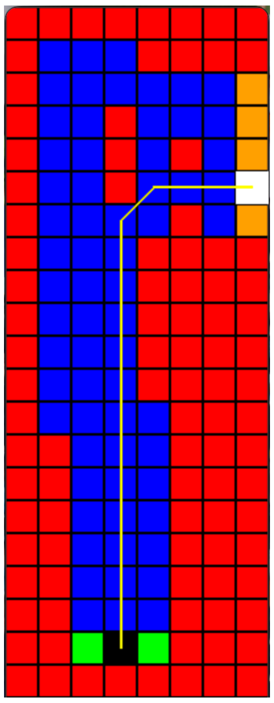
Optimal Trajectories:

The Complete visualisation of the car taking steps can be seen by running the code. Black box represents the start position. Rest of the colour scheme is same as the problem pdf.

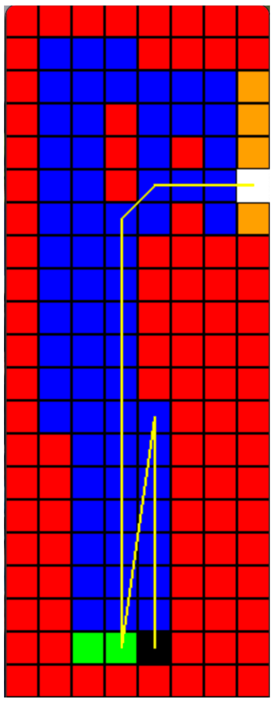
On-Policy Monte Carlo:



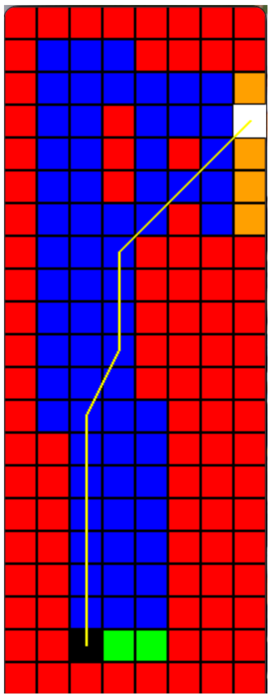
1ST START POSITION



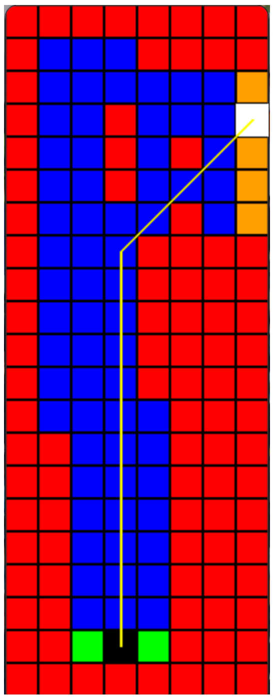
2ND START POSITION



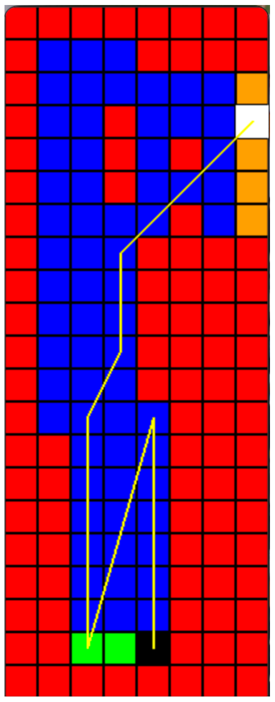
3RD START POSITION
(HIT WALL TO RESET)



1ST START POSITION

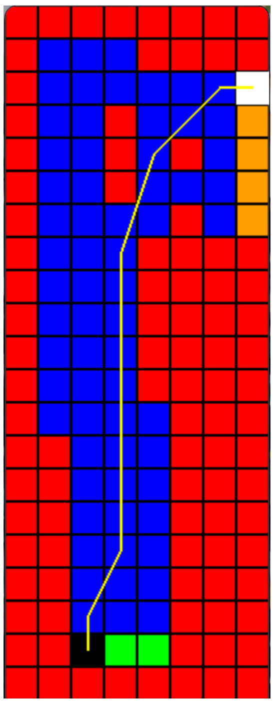


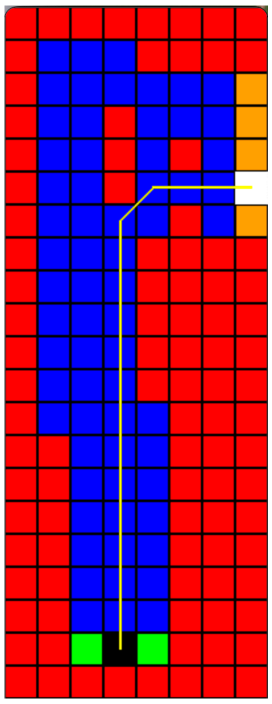
2ND START POSITION

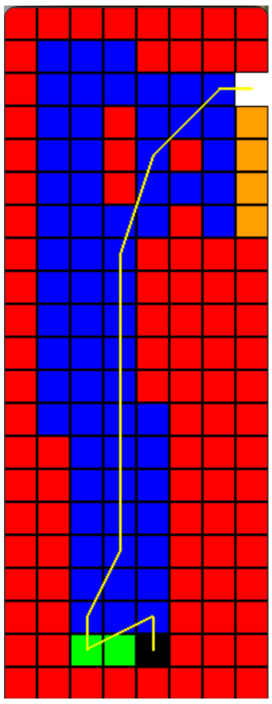


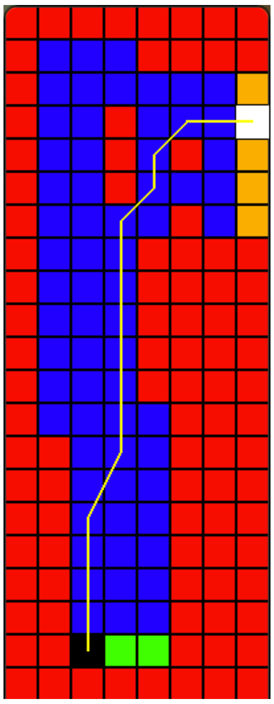
3RD START POSITION
(HIT WALL TO RESET)

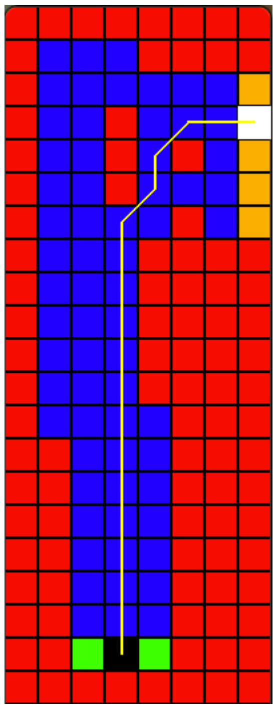
Off-Policy Monte Carlo:

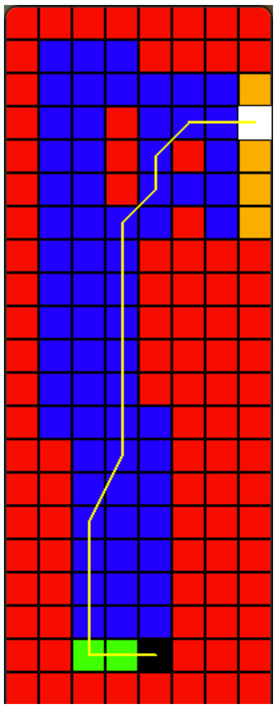

1ST START POSITION


2ND START POSITION


3RD START POSITION
(HIT WALL TO RESET)


1ST START POSITION


2ND START POSITION

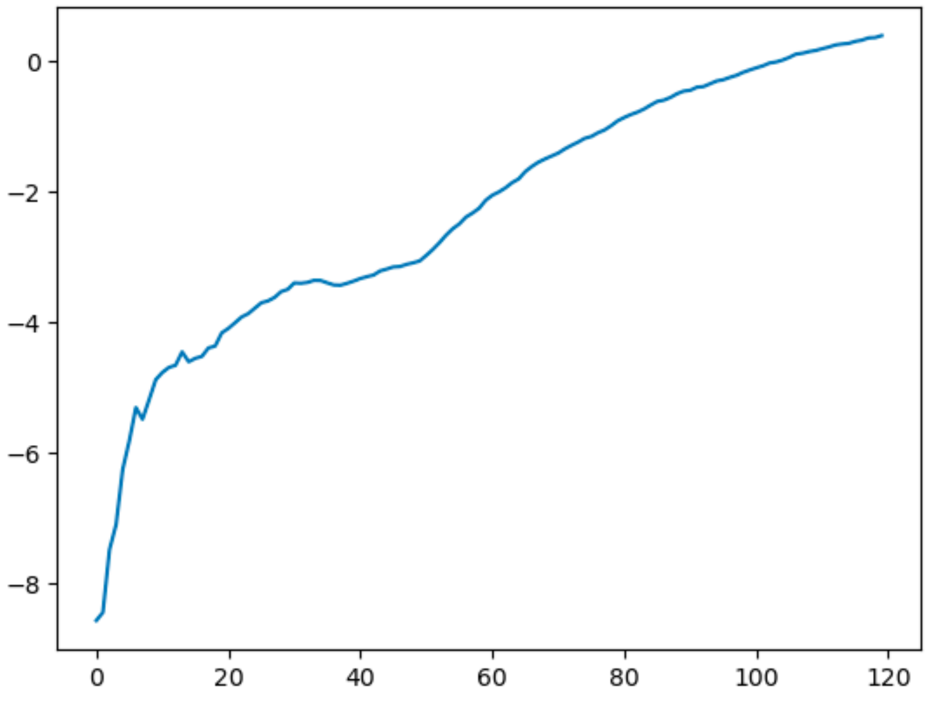

3RD START POSITION
(HIT WALL TO RESET)

Convergence:

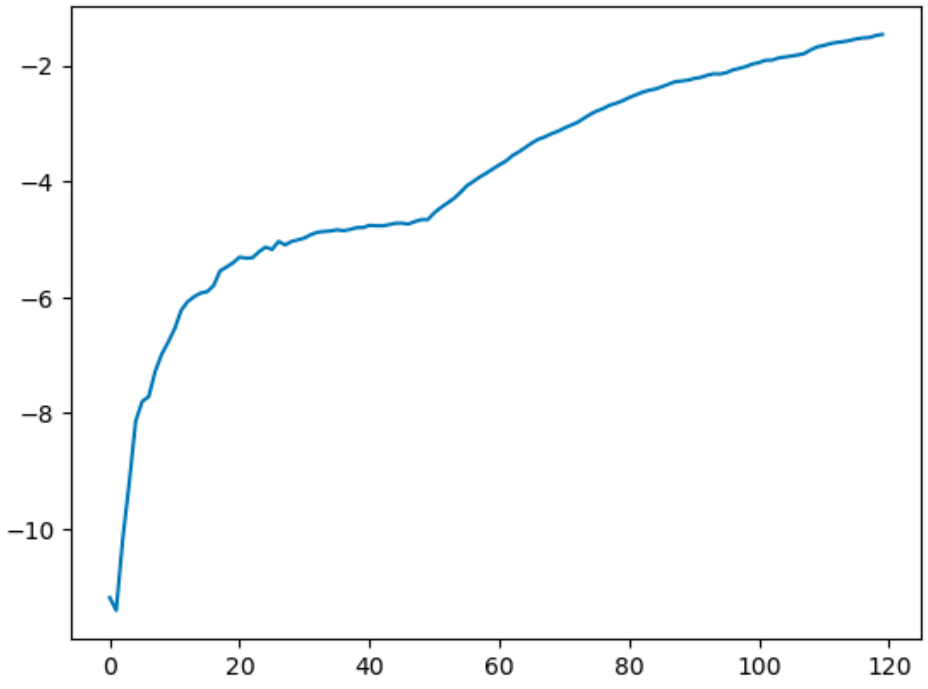
In this section I show the trends of state action values of two states over the 120 episodes:

- "0201005" = position: (2,1), velocity = (0,0), action = (0,1)
- "0301005" = position: (3,1), velocity = (0,0), action = (0,1)

<u>**On-Policy Monte Carlo:**</u>

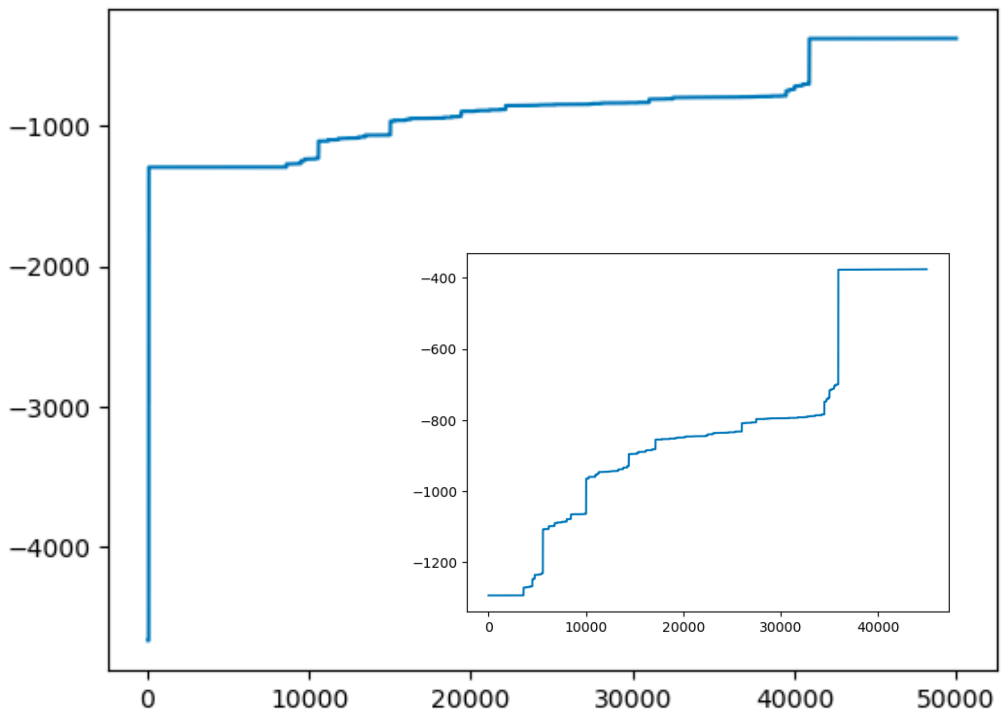


0201005

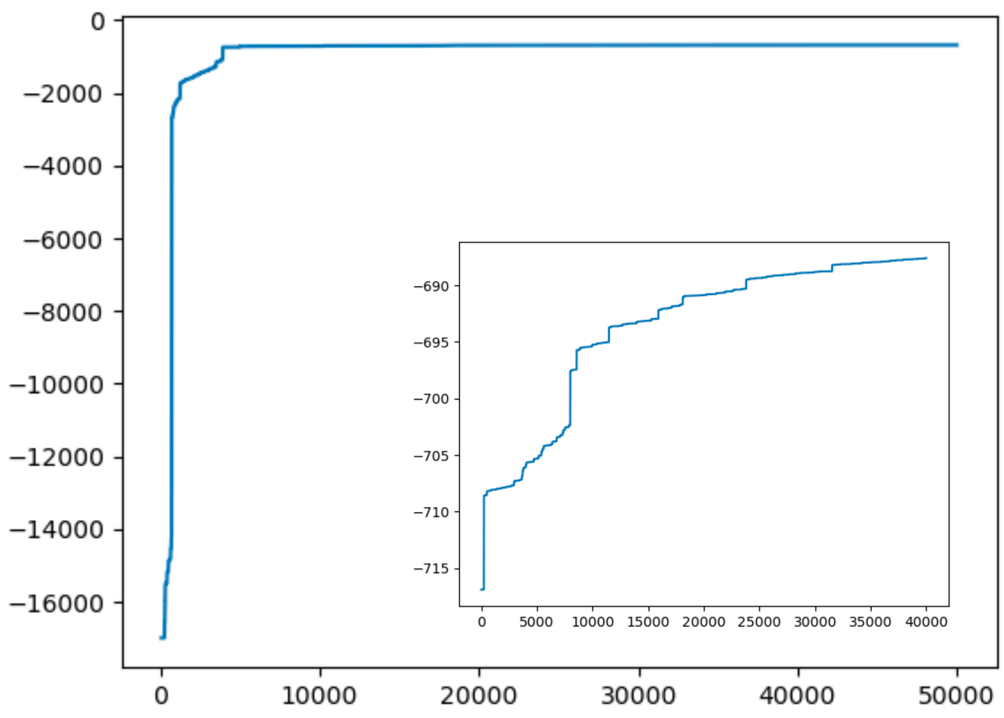


0301005

Off-Policy Monte Carlo:



0201005



0301005

Note: The smaller graphs are the zoomed in versions for the seemingly flat portions of the bigger graphs.

Some Observations:

- **For On-Policy:**
  - The epsilon value is changed at 51st episode, which is clearly visible in the graph.
  - This is because with epsilon 0.6, the state-action values starts reaching a saturation point after which a lesser epsilon again helps in converging further.
  - This is the reason I reduced the epsilon value so the exploration decreases in further episodes.

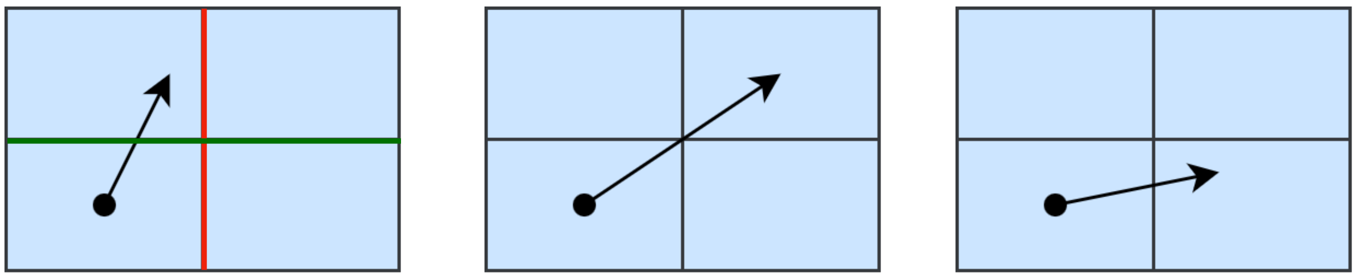
- **For Off-Policy:**
  - Although it may seem unnatural that there are sudden increases in the state-action values, but that is not the case.
  - This is happening due to the way we are updating the values.
  - Sometimes the values are not getting updated because the relative probability of the trajectory becomes 0 and in terms of code, the update loop breaks.
  - So when the value finally gets updated, the change is much more. Because that change has been piling up for all the episodes for which the value was not updated.

Now, even with the non-smooth graph of off-policy algorithm, it is pretty clear that off-policy algorithm is converging much slower than on-policy algorithm. In case of off-policy algorithm, the value keeps increasing by significant amounts for the whole 50000 episodes, whereas for the on-policy part it reaches convergence in order of 100 episodes.

Intuitively, this observation is valid as On-policy Monte Carlo algorithms should generally converge faster. This is because on On-policy methods, behaviour and target policy is same so we keep optimising our target policy while applying it in the episodes. This gives us the state-action values for the target policy and when those values are used to find the greedy action , our target policy becomes better. Whereas in the off-policy methods our behaviour policy is different that the target policy. We get state-action values for episodes run on the behaviour policy and not the target policy, so when we get greedy actions based on these values, the optimisation is slower because the values are not based on the target policy.

Collision Detection Algorithm:

In this section, I give a brief description of how I detected collision with walls or finish line in my algorithm. The algorithm works on the simple observation that a line going through a grid can do one of the three following things (as its velocity is always positive):



Now, There is a very elegant way of checking which of these cases is happening.
If the line crosses the red vertical before it crosses the green horizontal then case 3 occurs, if the line crosses the green horizontal before the red vertical, then case 1 occurs, if both are crossed together then case 2 occurs. Thus, by applying this rule in a serial manner we can determine which boxes the line intersects.