AE 4350 Final Project:
Reinforcement Learning for Path Finding for CubeSats

Pol Francesch Huc

08/31/2020

Contents

INTRODUCTION

Background

The purpose of this project is to generate a path finding algorithm for my research endeavors at my home university. There, I was part of a team building CubeSats to do obstacle avoidance maneuvers in a friction-less table.

A CubeSat is a small satellite that is composed of different units - or U's - which each measure 10x10x10 cm. A 3U CubeSat for example, would measure 30x10x10 cm and is practically composed of 3 1U CubeSats stacked. These small satellites allow for cheap ways to test technologies in space. In our case, we are working with a 3U Cubesat, however this is easily modifiable in the software.

In order to test satellites before launching them, many companies and government agencies have friction-less surfaces. These are made of polished granite that is chemically treated, and produces a negligible amount of friction.

Our CubeSat is equipped with two cold gas thrusters to be able to move in two directions, and a reaction wheel which allows it to spin. It is sitting on top of a levitating surface, which blows air down onto the friction-less table to generate lift. This surface is roughly circular, and is somewhat larger than the CubeSat.

For communications with the ground station, our CubeSat uses laser technology, which requires line of sight to operate. If an obstacle is to come between the CubeSat and the ground station, the CubeSat will lose the communication link. This is not much of an issue, as it will continue past the obstacle and then re-synchronize with the ground station.

Generating Scenarios

In order to make sure the algorithm is versatile, we need to be able to create many obstacle maps. For these, we can specify the size of the map, the obstacles, and the resolution of the map. The algorithm chosen breaks the map into nodes, the resolution parameter specifies how many nodes (or cells, if we wish to talk about matrices) per meter we will have.

I am generating maps based on given size parameters, and then randomly adding the start, end locations and obstacles. Since the randomization is not perfect, and it is possible that some maps are just not desirable to be tested (ie obstacles are all small so it would make for a boring test, or the path is completely blocked so it would be impossible), the user is able to select a map at the beginning of their session.

Note that for these maps, the yellow square at the top is the start state, the green at the bottom is the finish area, and the black sections are the obstacles. See examples of maps below:

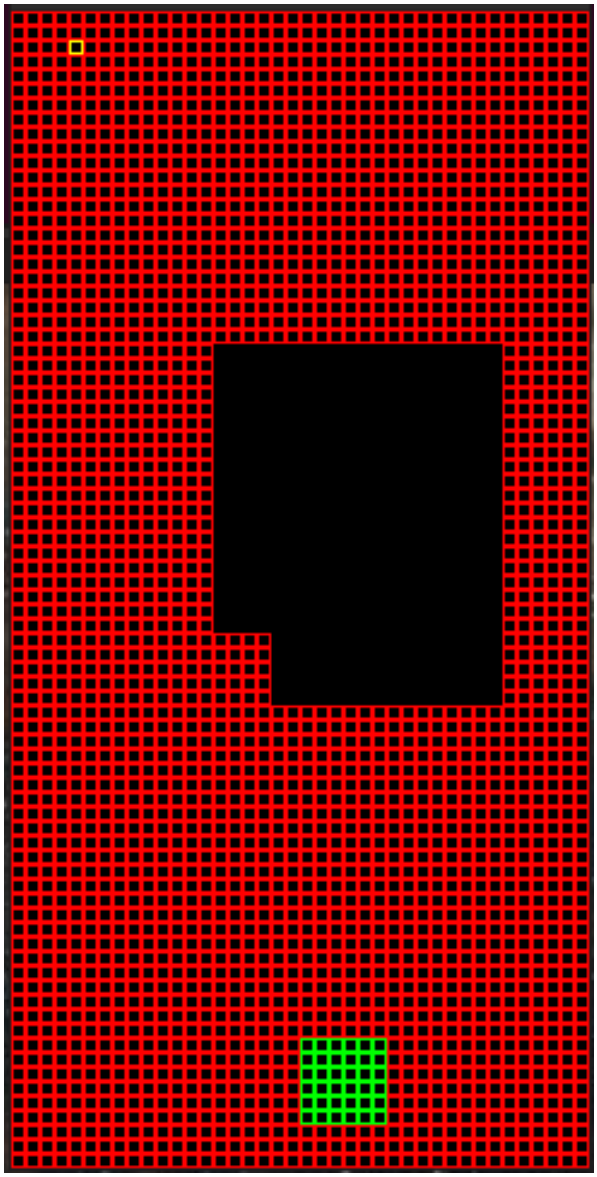

Figure 1: Example Map

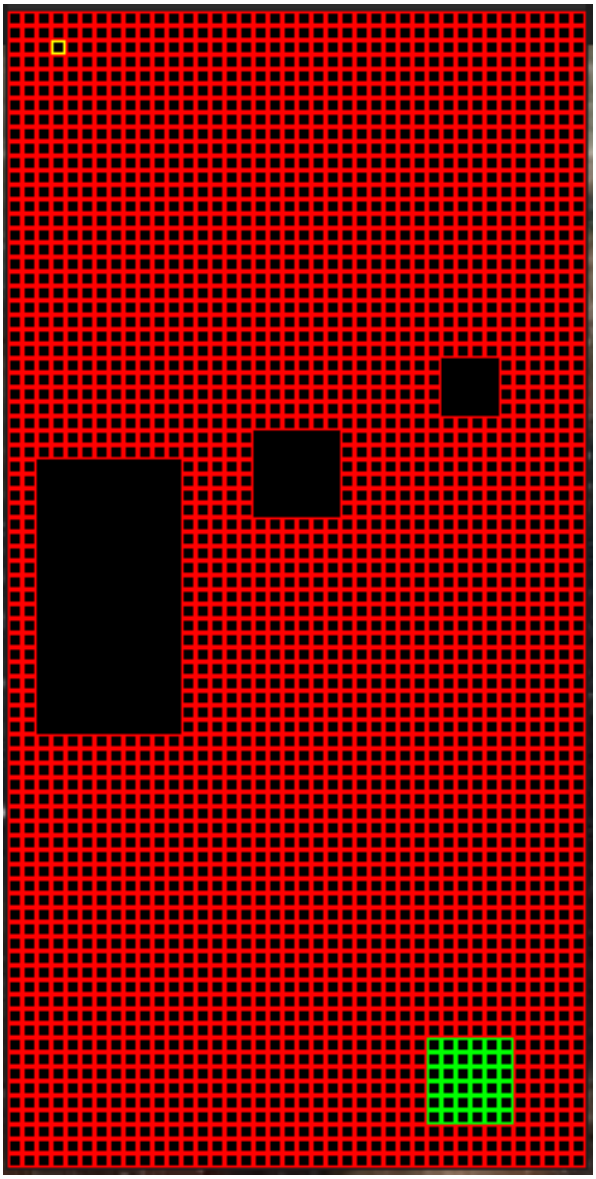

Figure 2: Example Map

THE RACETRACK PROBLEM

The racetrack problem is a common problem is reinforcement learning which is used as a learning tool for many. As this was my first experience with machine learning, I chose to adapt the model for the racetrack problem to my CubeSat, in order to simplify my implementation.

Problem Description

The racetrack problem I will be solving now is the one found in Sutton and Barto, the textbook which I followed in order to complete this project. In brief, a car is placed on a racetrack and it has to reach the end of the racetrack in as little steps as possible. Some example racetracks from the book:

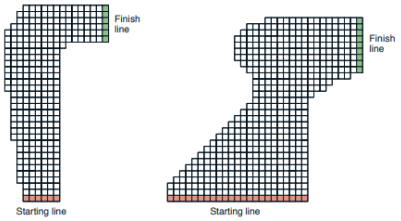

Figure 5.5: A couple of right turns for the racetrack task.

Figure 3: Example Racetracks (Sutton Barto)

Modelling Aspects

In this problem, the car begins with zero velocity and is simply tasked with reaching the end line as quickly as possible. If the car goes over the edge it must start over. At each step, the car has the option to take one of nine actions. The first of these nine is to do nothing. The other eight are just the possibility to increase speed in the eight directions (north, northwest, west, etc). This gives us nine possible actions to take.

This is not always the case though as sometimes we are not allowed to speed up. This is due to the algorithm that we are using, which is a Monte Carlo Off-Policy Algorithm. Amongst other things, this algorithm has the requirement that you must be aware of all the possible states. If the car is allowed to always speed up, then the amount of possible states increases dramatically. On the other hand, if we set a speed limit, we can limit this number and it makes our calculations much simpler. It is also much more realistic, as our real car does have a maximum speed.

This model, evidently, has its limitations. The most obvious is that you can only accelerate in certain pre-set directions and that turning can be done on a whim. This portion is not too realistic, and will be slightly improved later on. It is also limiting that you must have previous knowledge of all possible states. This is more a feature of the Monte Carlo algorithm, however it does complicate our approach.

Monte Carlo Off-Policy Control Algorithm

As with all reinforcement learning of this type, we wish to find the optimal policy. Off-policy methods do this by using two policies at once. One, the behavioural policy, has to guarantee that it will sample all possible combinations of states and actions, meaning it must be soft. Additionally we have our target policy. The thinking behind this framework is that since we have our behaviour policy, our target policy can be deterministic, and does not need to have any effects of randomness.

In our case, the target policy is greedy, meaning it takes the best action every time, and our behaviour policy is ϵ -greedy. This allows it to take good actions, whilst still being practically guaranteed to check all possible states.

I followed the pseudo-code found in Sutton and Barto in order to get the update of the policy.

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 4: Off-Policy Monte Carlo Control Algorithm Pseudo-Code (Sutton Barto)

The variables used are defined below:

π	=	Policy used to decide what to do at each state
π_*	=	Optimal policy
ϵ	=	Probability that the behavioural policy will choose a random action
γ	=	Learning Parameter
s	=	State
a	=	Action
R	=	Return for taking an action at a given state

Q = Action-value function

C = Cummulative sum of the weights given so far

Results

The algorithm and model were implemented following Rastogi's work which is available on their GitHub. The algorithm was tested on the following racetrack. It was chosen due to the small finish line, and number of turns.

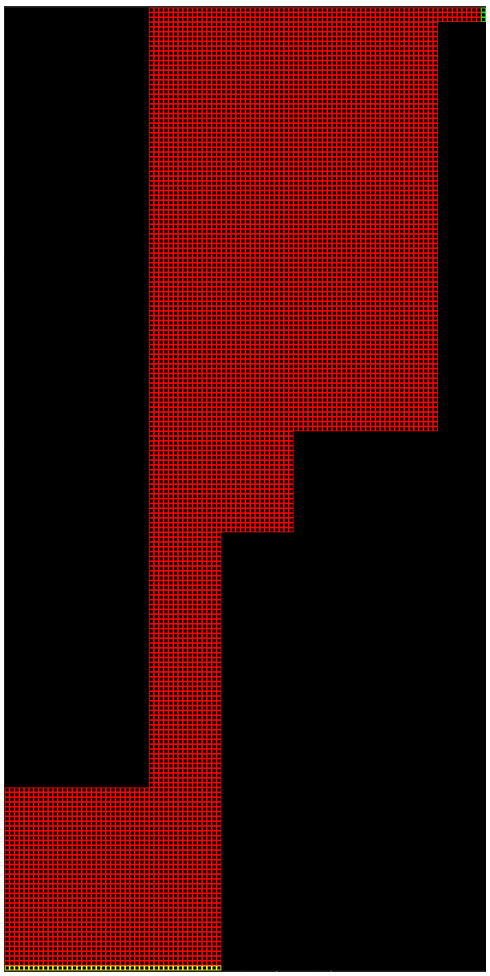

Figure 5: Racetrack used to test the algorithm

From running the code, I was able to get interesting results as it improved over time. As expected the solution takes some time to reach the optimum, but once it does, it works rather nicely. This is demonstrated by the plot below. Please keep in mind that the total rewards are negative as the agent receives a negative reward for every step it takes, and reaching the finish does not give it a reward at all - it simply ends the execution.

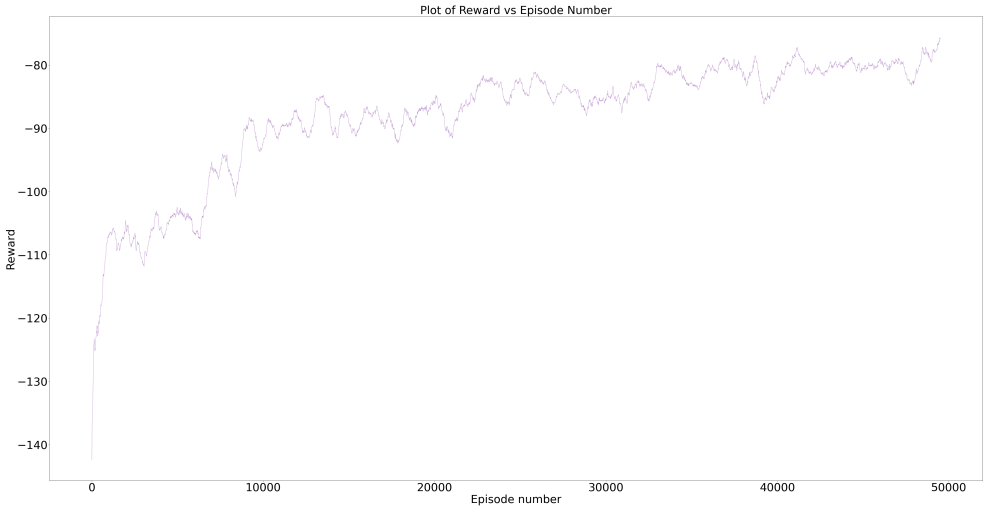

Figure 6: Results Racetrack

I will not be performing sensitivity analysis on this problem, as I can simply do so on our actual model.

CREATING A NEW MODEL FOR CUBESAT

The model for the car and racetrack is sufficient to get a good idea of how a car would behave in such an environment. That being said, it certainly needs some work to be applied to my use case with CubeSats and fricitonless surfaces.

Changes in the Agent

In order to act more like the CubeSat the agent must be changed. First, we must change the actions that can be taken. Although the actions for the racetrack were very nice as they were simple to reference, they are not sufficient to model the CubeSat. From a high level, the CubeSat has 8 possible actions. It can still do nothing, but now it can turn on and off its thrusters (it has two, one in the front and one in the back), and spin the reaction wheel left, right or stop it. From these we get our eight actions.

Unfortunately, these actions are much more difficult to model and hence restrictions are applied. As mentioned before, the states must be known beforehand, and hence we must restrict the turning of the reaction wheel such that it can only turn in increments of 45 degrees (much like before). But now, the thrusters are aligned, so the CubeSat will only thrust in two directions. If it is facing forward and wants to go left, it will first have to turn.

There is one complication that comes from this however. As with the racetrack, I apply a speed limit both in the positive and negative direction, however this time our speed limit is much more easily passed. If our CubeSat turns on the thrusters, it could choose to do nothing for a while and leave them on, which would make it breach this speed limit. I must hence enforce more stringent requirements when determining what actions are possible to be taken. Sometimes, I must even remove the do nothing action, to force the agent's hand and make sure it does not step out of the bounds of our problem.

Changes in the Environment

The environment must also be changed. In this case, we must chance our state space representation of the system. Before, the state space simply held information about the position and velocity. From there, the algorithm would decide what the best action to take is based on the current state.

Now, I must add some more information. First is the angle, this is obvious as it is turning, so it is of course necessary. It must also have information about the thrusters and the reaction wheel. It must store whether each thruster is on or off, and whether the reaction wheel is spinning left, right, or not at all. This affects the current condition and what the best next action, and therefore it must be passed on as information for the algorithm. Although these are not seemingly many options, it adds a lot of overhead in calculation time as these matrices with which the information is stored become sluggish, to say the least.

With these changes applied however, we are now ready to model our system.

Sample Solution

The algorithm was tested on the following map:

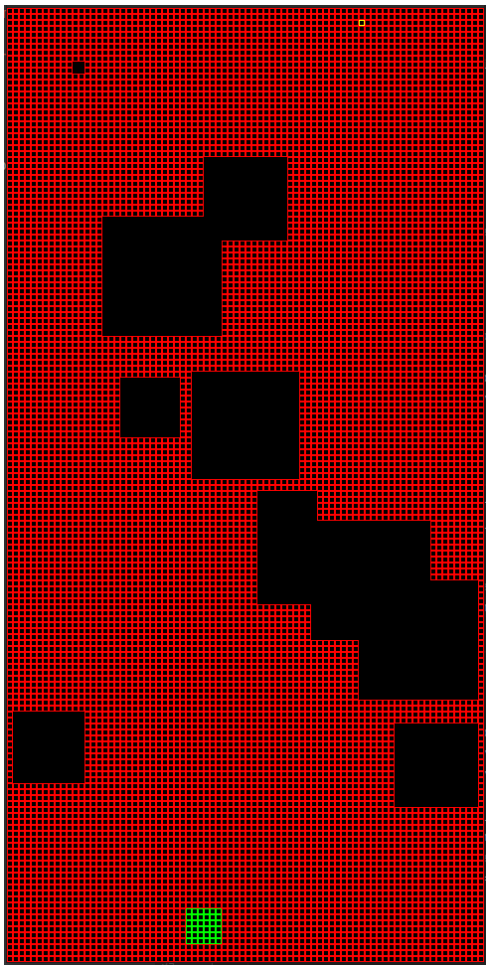

Figure 7: Map through which the CubeSat runs

Although it is rather small, the increase in complexity of the state space model heavily delays progress in this case. Having a bigger map would have made the whole algorithm take far too long to finish its training - this map takes about 90 minutes on my machine.

The results from this can be seen below:

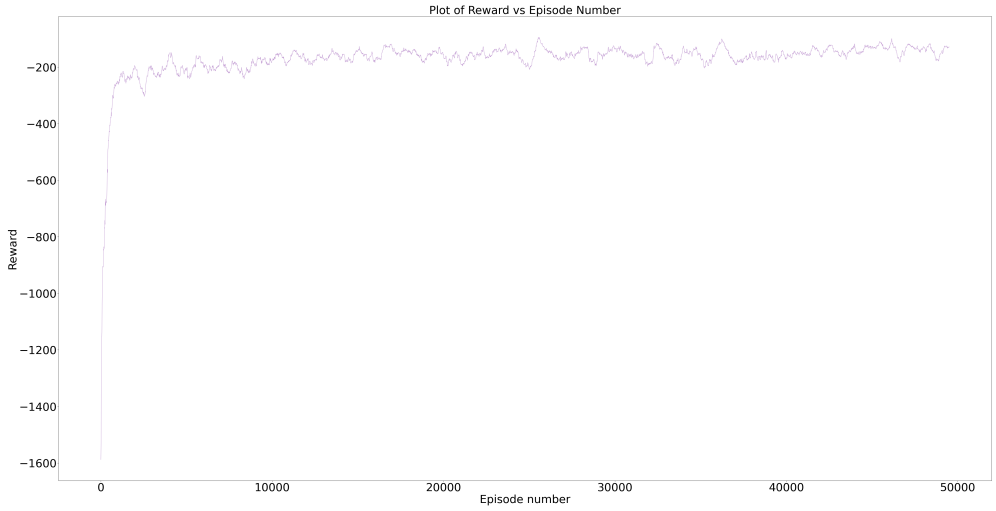

Figure 8: Episodes vs rewards ($\epsilon = 0.1$)

Discussion of Solution

We can see from this learning process, that it gets to the optimal policy much more quickly than before. This makes sense though, as before our plot was larger, and our end state was harder to reach. It seems that this algorithm learns quite quickly, and that within the first 10,000 episodes, it has already approximated the optimal policy.

From this, it is worth noting a couple of things. First, the way I am handling rewards and termination of episodes is different. Before, with the racetrack, we were running each episode until they reached the finish line. This meant that the first 50 episodes were quite slow, but once the algorithm got the idea, it went pretty quickly. In this case however, it takes much longer than 50 episodes as there are many more states. Because of this, I decided to terminate the episode if it went over 100,000 steps. Logically, this should be more than enough to get to the destination. This is why the two plots (racetrack vs map) look so different from one another.

SENSITIVITY ANALYSIS

We perform a simple sensitivity analysis on the greediness of our algorithm.

Method

My methodology is quite simple. I take a sample map which the algorithm has never seen, and hence has no previous data to use as a stepping stone, and run the algorithm with different ϵ values

Sample Solutions

This test was done on the following map:

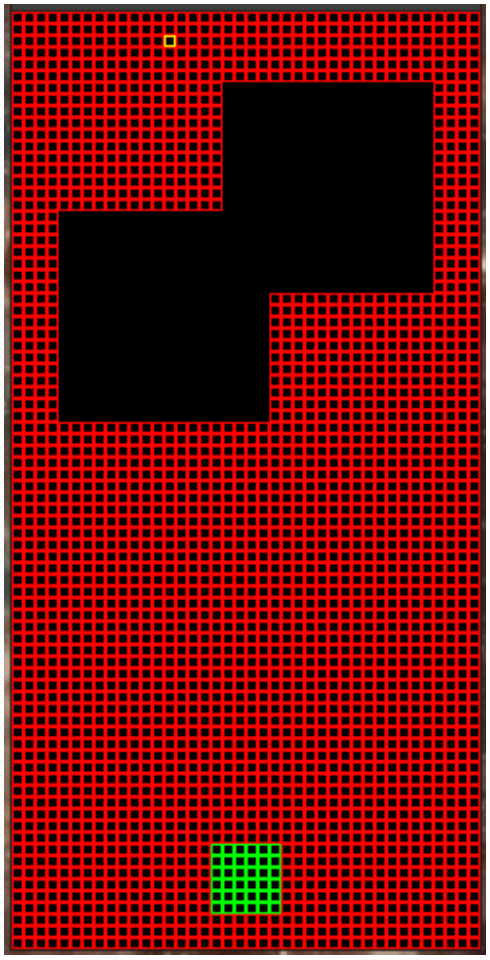

Figure 9: Map through which the CubeSat runs

The different greediness can be seen having an effect on the set of plots below.

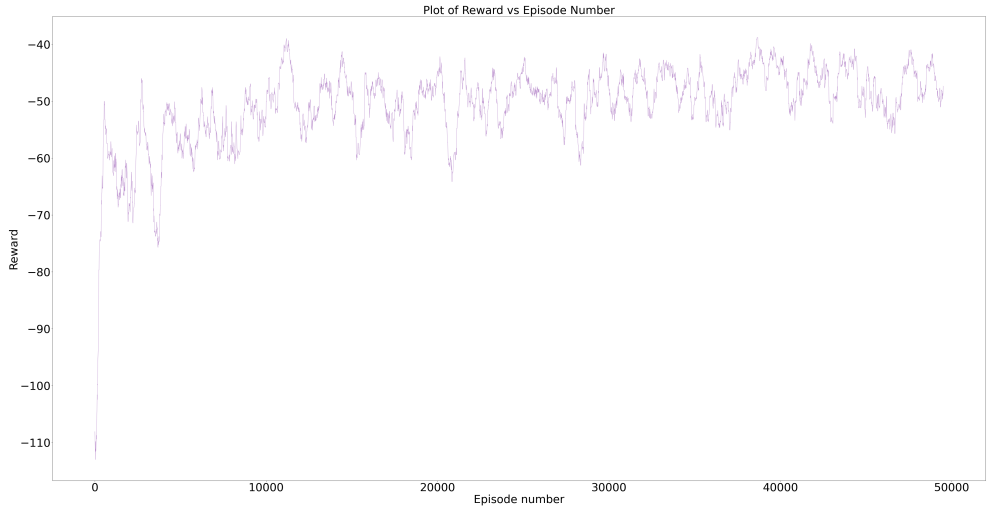

Figure 10: $\epsilon = 0.1$

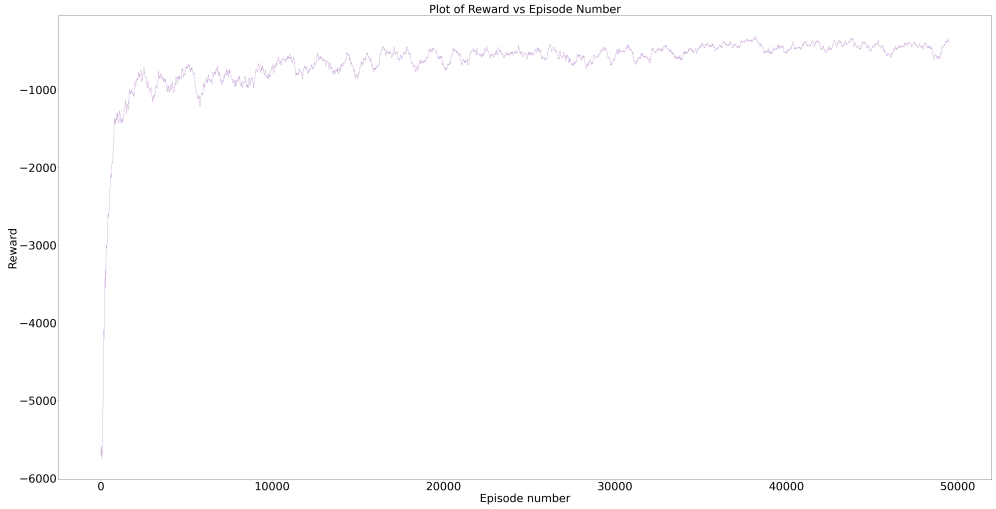

Figure 11: $\epsilon = 0.25$

Discussion of Solution

These plots show some interesting results when it comes to varying the probability of random action, ϵ. At first glance, it seems that the solution given by the higher ϵ is better, as it returns a smoother reward per episode, however this may not be the case. In truth, using a higher ϵ is not beneficial at all and actually reduces accuracy and increases time of completion.

First, I note that the reward for $\epsilon = 0.1$ stabilizes at approximately -50, whilst for the case of ϵ it is closer to -900. Clearly this is much worse. One may think that it does have a bright side, since it is smoother, but this is simply a visual effect. The peaks and troughs in the $\epsilon = 0.1$ case are of an amplitude of -30 at the most. This is insignificant when the scale for $\epsilon = 0.25$ is near the thousands. In fact, due to the higher randomness, it is likely that it would be a lot less smooth.

Second, although this is not notable in the plot, it takes much longer to run the code with $\epsilon = 0.25$ than with $\epsilon = 0.1$. The latter took a little less than ten minutes, whilst the former was at least three times that.

It seems then that having $\epsilon = 0.1$ is best in this case. I attempted running at a lower epsilon, but it struggles a lot as it has a much lower chance of getting to all the states, and hence has issues reaching an optimal policy. Obviously, this would be much smoother with the target policy, which has no chance of taking a random action, and we would expect much more consistent results.

One final note is that it may be worth investigating the use of a diminishing epsilon in this code. Although this would be more useful for an on-policy algorithm, it would make our behavioural policy more reliable.

CONCLUSION

The point of this project, for me, was to improve on my previous attempt to solve this problem. In the past, I used an A* algorithm to solve this using MATLAB. Though certain areas have certainly been advanced, others are actually worse off.

To begin with the positives, this is a great improvement as the algorithm can also give me information about actions, and takes into account how it must turn to fire thrusters. A complicated model such as this one, was not possible for me to implement using the A* algorithm. Additionally, this Monte Carlo Off-Policy Algorithm has the added benefit that it must visit every state, hence the found policy is closer to the real optimal policy, making this solution much more accurate.

On the other hand, this algorithm is much slower to run. It can take tens of minutes to get a good policy. Compared to the mere seconds it takes the A* algorithm, this is clearly a drawback. This may be due to python being a quite slow program, but it is likely that the complexity of the state space model adds to this timing issue. Unfortunately, the latter is a needed requirement for Monte Carlo Off Policy Algorithms.

Though this code gives us an improvement over the A* algorithm in the model fidelity, it is still limiting and it comes at a great cost; it would not be possible to do live obstacle avoidance for example. It is likely that models that do not require the user to know all possible states would fare better here, and give better results.

RECOMMENDATIONS

From here, one may be tempted to simply revert back to the A* algorithm, however the action information is rather valuable. Hence, applying a different machine learning algorithm all together, may be worth the time as it will combine our path planning with a simple list of actions to take.

It would also be important to drastically reduce the time it takes to reach the optimal policy. From this information, it is fair to say that the algorithm would not be doing all the learning every time it got a map. It would be easier, and quicker, to get a path from the A* algorithm, and then use a machine learning algorithm to optimize the path, and give action information. This would combine the best of both worlds.

Looking further ahead, it would be beneficial to run this algorithm while the CubeSat moves. While it is nice for modelling, it is unlikely that in the real world the CubeSat would be aware of all the surroundings. Some things will be blocked from the CubeSats vision, and others will have sensor reading errors as they are partially obscured. In both cases, the CubeSat would need to reevaluate it's path.

BIBLIOGRAPHY

Sutton, Richard, and Barto, Andrew. "Reinforcement Learning: An Introduction." 2nd Edition. MIT Press, 2018.

Rastogi, Aditya. "Solving Racetrack in Renforcement Learning using Monte Carlo Control." Medium, towardsdatascience.com, 6 Jan 2020. URL: https://towardsdatascience.com/solving-racetrack-in-reinforcement-learning-using-monte-carlo-control-bdee2aa4f04e

APPENDIX: READ ME FOR CODE

I am including two codes in this report. The first is the one from Rastogi - though slightly modified to better meet my needs -, and the second one is my implementation of their code, which is heavily modified. The following subsections are made for running the latter.

Libraries

To run this code I am using the newest version of Python 3 along with some libraries:
numpy - for array math
matplotlib - for plotting
tqdm - for a progress bar when running RL
imageio - for creating gifs
pygame - used for the map

Running Code

I developed the code with the PyCharm IDE, however it is not recommended to run it with that. This is as the PyCharm IDE can cause issues with Matplotlib. It is better to just run the code from the terminal. You must only run the main.py file, as the rest are simply helper files.

Once you have run the code, it will first ask you to select a map. It does this by showing you maps until you find one you like. Once it shows you a map, the code stops. You must press space bar while selecting the map window, and then the terminal will have you input whether you like this map, and hence perform the algorithm on it, or would like to see other maps.

After you have selected your map, sit back and relax as it takes a while to run. Please note that although tqdm is quite good, it does not give very accurate estimates at first. So even if it says it will take 200 hours to complete, know that on my desktop PC usually takes around twenty minutes, though this of course depends on your settings and hardware. Finally, the code will end, and you will find two new plots in the images directory.

Options

There are some options found in the main.py file which can easily be changed if you would like to check versatility. The first two are the size of the map and the resolution. I would keep the size values between 2 and 6, anything smaller and the rudimentary math I used to create maps begins to break down. The same is true for anything bigger. The resolution can be changed at will, though it is not a very useful metric.

The next important value is the cell edge. This is used by pygame to determine the size of the window. If you cannot see the full size of the window, reduce this value.

Finally, if you would not like to remake graphs, you have the choice to do this. You can also rename your images/plots such that they will not overwrite each other.

There is also the option to change the ϵ value.

Rastogi's Code

This code is significantly faster to run the set up as there are not as many states. From our end, the procedure is somewhat similar. You can simply run racetrack_problem.py and it will run exactly the same racetrack I have shown in the report.

If you would like to get a random racetrack, you must simply change the options in line 646. Note that if you have already run this code, and you re-run it, it will overwrite save files and the plots.