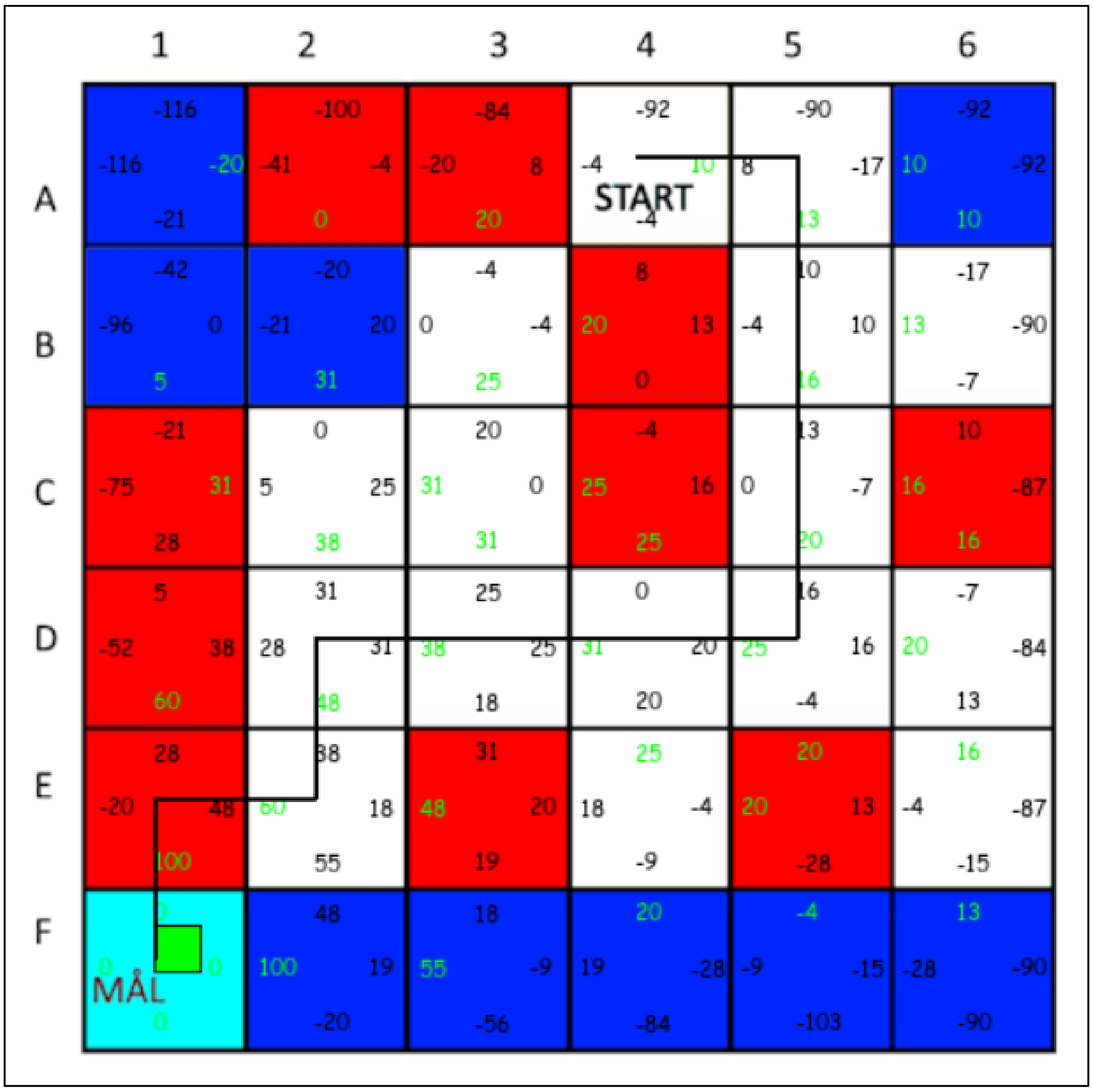
Q-learning

Mathias Eira Karlsen

DET-2602 24H Introduksjon maskinlæring og AI

13 October 2024



1 INTRODUCTION

A robot must explore unknown terrain. In this report I'm going to show how Q-learning can be used to find the safest path through the terrain. And how Q-learning does when compared to just randomly exploring the terrain. I am also going to show how tweaking the learning parameters can affect Q-learning.

The terrain the robot is going to explore I split into 36 states as shown in the map below.

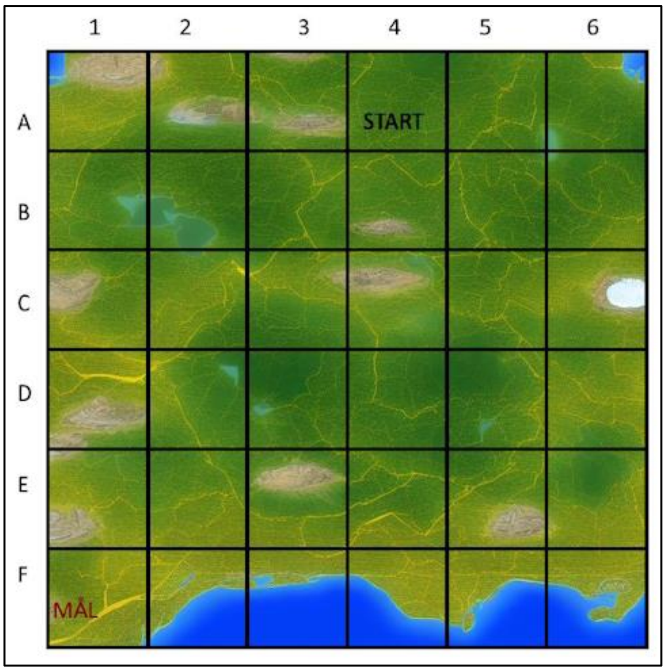


Figure 1 -- Map of terrain[1]

The map has two types of obstacles, steep terrain and water. So, the map can be further simplified like this, where red squares are steep terrain and blue squares are water.

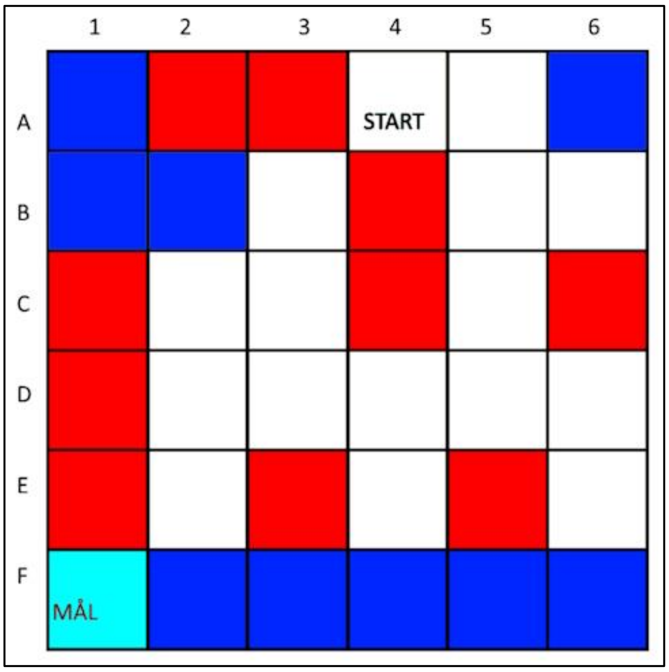


Figure 2 -- Simplified map of terrain[1]

2 THEORY

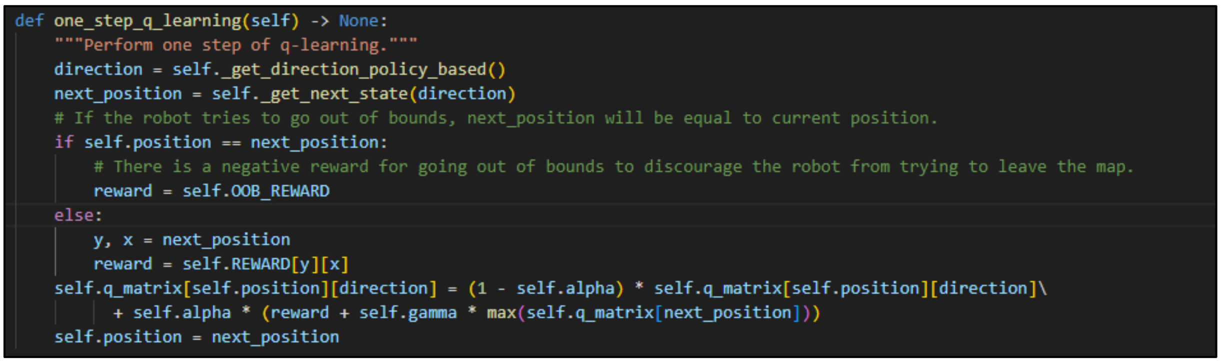
The robot has a simple API that allows it to move up, down, left and right. There are two main algorithms that will be used to control the robot: Monte Carlo-simulation, where the robot moves randomly until it reaches the goal. And Q-learning, where the robot learns how to safely navigate the terrain.

Q-learning is a reinforcement learning algorithm to learn the value of actions on a state. Every time the robot does an action the Q-matrix of the robot gets updates using this formula: $Q^{New}(S_t, A_t) \leftarrow (1-\alpha) \times Q(S_t, A_t) + \alpha \times (R_{t+1} + \gamma \times maxQ(S_{t+1}))$ [2]

- S_t is the state before and action is done.
- S_{t+1} is the state after the action
- A_t is the action.
- Alpha (α) is the learning rate.
- R_{t+1} is the reward for going to the new state.
- Gamma (γ) determines how important future rewards are to the robot.
- maxQ(S_{t+1}) is a function that gets the highest reward in the Q-matrix for state S_{t+1}

The formula can be split into two parts. Old information: $(1-\alpha) \times Q(S_t, A_t)$ and new information: $\alpha \times (R_{t+1} + \gamma \times maxQ(S_{t+1})$. The learning rate determines how much these parts are weighted. With a learning rate of 0 the Q-matrix doesn't change, and the robot will not learn anything new. With a learning rate of 1, only the new information is kept, and the old information is discarded.

When running Q-learning the robot is placed somewhere random on the map. It will then choose an action based on its policy. Every time it does an action, the Q-matrix is updated using the formula. The position gets updated and If the robot reaches the goal, it will be placed randomly again, and the algorithm continues. Every time the robot goes from being placed to reaching the goal, that is called an episode or epoch. The algorithm can be run for a certain number of epochs or until the Q-matrix converges, which is when the Q-matrix no longer gets changed.



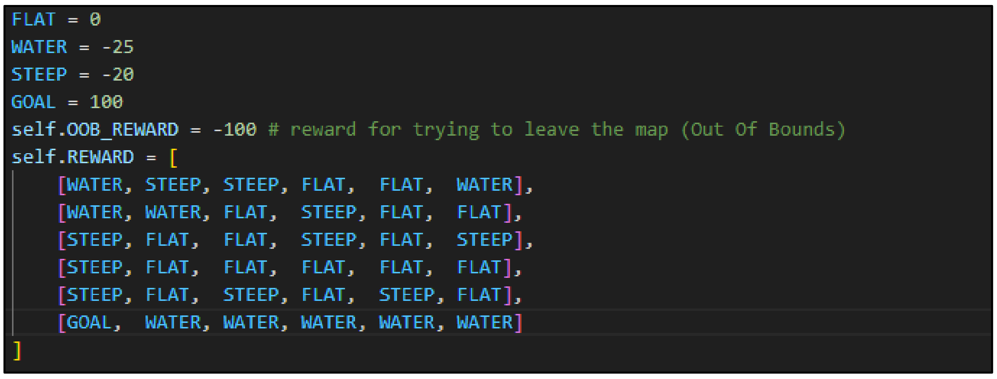
```python
def one_step_q_learning(self) -> None:
    """Perform one step of q-learning."""
    direction = self._get_direction_policy_based()
    next_position = self._get_next_state(direction)
    # If the robot tries to go out of bounds, next_position will be equal to current position.
    if self.position == next_position:
        # There is a negative reward for going out of bounds to discourage the robot from trying to leave the map.
        reward = self.OOB_REWARD
    else:
        y, x = next_position
        reward = self.REWARD[y][x]
    self.q_matrix[self.position][direction] = (1 - self.alpha) * self.q_matrix[self.position][direction]\
          + self.alpha * (reward + self.gamma * max(self.q_matrix[next_position]))
    self.position = next_position
```

Figure 3 -- Q-learning algorithm. Python code

To determine which action the robot will pick it will use a policy-function. There are two ways the robot can choose how to move. The first is based on Monte Carlo, where the robot will pick a random direction. The second is based to Epsilon-greedy, where the robot will pick the direction that has the highest value in the Q-matrix.

3 METHOD

In the map, there are 4 types of states the robot can be in. White squares are flat terrain, blue squares are water, red squares are steep terrain, and the bottom left corner of the map is the goal. Each type of terrain has an associated reward. I decided to pick 0 for flat terrain. -20 for steep terrain do discourage going on more risky uneven terrain. For water I decided to set the reward to -25 because it takes a lot more energy and there could be unknown risks like sea creatures. The reward for reaching the goal is 100. It's high to encourage the robot to reach the goal even if it must take some risk to get there. But not so high that the robot completely ignores all risks to reach the goal as fast as possible. To discourage trying to leave the map, the reward for trying to leave the map is -100. In the code the robot has a 2 by 2 reward matrix where it can use its coordinates to get the reward for a state.



```
FLAT = 0
WATER = -25
STEEP = -20
GOAL = 100
self.OOB_REWARD = -100 # reward for trying to leave the map (Out Of Bounds)
self.REWARD = [
    [WATER, STEEP, STEEP, FLAT,  FLAT,  WATER],
    [WATER, WATER, FLAT,  STEEP, FLAT,  FLAT],
    [STEEP, FLAT,  FLAT,  STEEP, FLAT,  STEEP],
    [STEEP, FLAT,  FLAT,  FLAT,  FLAT,  FLAT],
    [STEEP, FLAT,  STEEP, FLAT,  STEEP, FLAT],
    [GOAL,  WATER, WATER, WATER, WATER, WATER]
]
```

Figure 4 -- Reward matrix

For each state the robot is in, it has 4 choices, it can move up, down, left, or right. So, the Q-matrix is stored as a dictionary where the key is a tuple of the coordinates for a state, and the value is a list with 4 values representing the learned value of the 4 choices. When the robot is initialized all the values are 0 so the Q-matrix might look something like this: {(0, 0): [0, 0, 0, 0], (0, 1): [0, 0, 0, 0], (1, 0): [0, 0, 0, 0], (1, 1): [0, 0, 0, 0]}. But after running Q-learning for a few episodes the matrix will have some values and might look like this: {(0, 0): [-100.0, -25.0, -100.0, -36.0], (0, 1): [-100.0, -25.0, -25.0, -20.0], (1, 0): [-25.0, -20.0, -100.0, -25.0], (1, 1): [-20.0, 0.0, -25.0, 0.0]}.

Gamma determines how important future rewards are. Because there is only one space that has a positive reward, it is important for the robot to reach that space. Since the path to the goal can be long, I decided to set gamma to 0,8. This way the reward of the goal can have a large influence on all the states on the map.

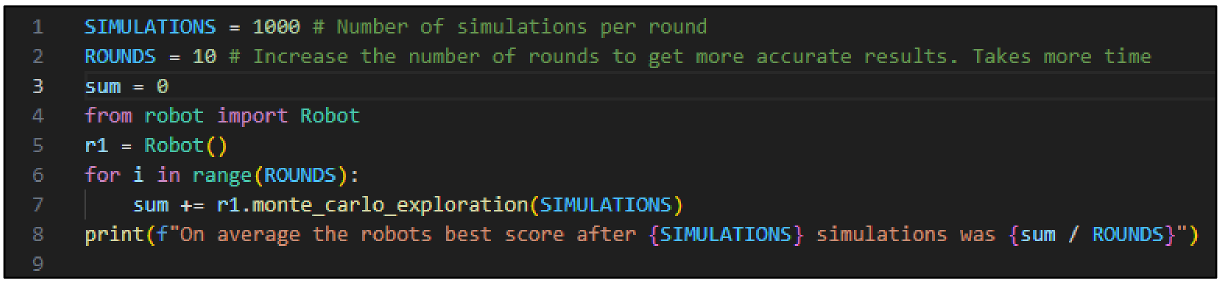
The learning rate alpha is also very important for how the Q-learning algorithm behaves. Since the map is fully static and deterministic, I decided to set the learning rate to 1. If you think about a simple map with just two states, a start and a goal, with a learning rate of 1 the robot would only have to go from the start to the goal to set the correct Q-value for that action, but with a lower learning rate, it will eventually converge at the same value, but it would take much longer. The same idea applies to a bigger map. A larger alpha leads to faster convergence.

For choosing where to move, the robot has a very simple policy. 50% of the time it will use Monte Carlo, and 50% of the time it will use epsilon-greedy. This gives the robot a mix of exploration and exploitation. It will exploit what it currently knows but the randomness will make sure it never gets stuck and that it will eventually explore all its possibilities.

The robot has 3 main functions to explore the map. The first is Monte Carlo-exploration, where the robot only moves randomly, and counts the rewards it collects. The function runs for a certain number of epochs, and then returns the highest reward it was able to collect in a single epoch. The second function is Q-learning, where it runs Q-learning for a certain number of epochs, updating the Q-matrix. The last exploration function is Q-learning converge, where the robot runs Q-learning until the Q-matrix doesn't get updated for a certain number of epochs.

After Q-learning has been used, the best path from start to goal is created by setting the robots position to start and running epsilon-greedy until it reaches the goal.

To calculate statistics on how Monte Carlo and Q-learning performed, a program that does many simulations, counts how the robot did, and then calculates an average was used. This is the program for calculating the average score for Monte Carlo, the same thing was done for Q-learning.



```python
SIMULATIONS = 1000 # Number of simulations per round
ROUNDS = 10 # Increase the number of rounds to get more accurate results. Takes more time
sum = 0
from robot import Robot
r1 = Robot()
for i in range(ROUNDS):
    sum += r1.monte_carlo_exploration(SIMULATIONS)
print(f"On average the robots best score after {SIMULATIONS} simulations was {sum / ROUNDS}")

```

Figure 5 -- Average reward Monte Carlo-exploration. Python code

4 RESULTS

It takes a lot of simulations to find a safe path while traveling randomly. Most of the time the robot will get a very negative reward. Even with over 100 simulations, most of the time the best reward is still negative

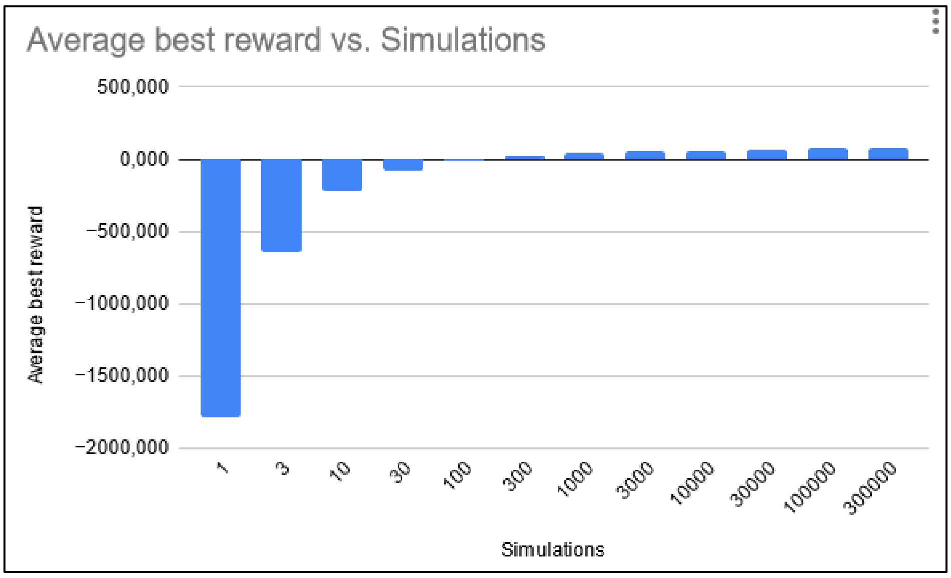


Figure 6 -- Graph of Average best reward after n simulations Monte Carlo-exploration

By removing the negative reward simulations, you can see just how many simulations it takes to

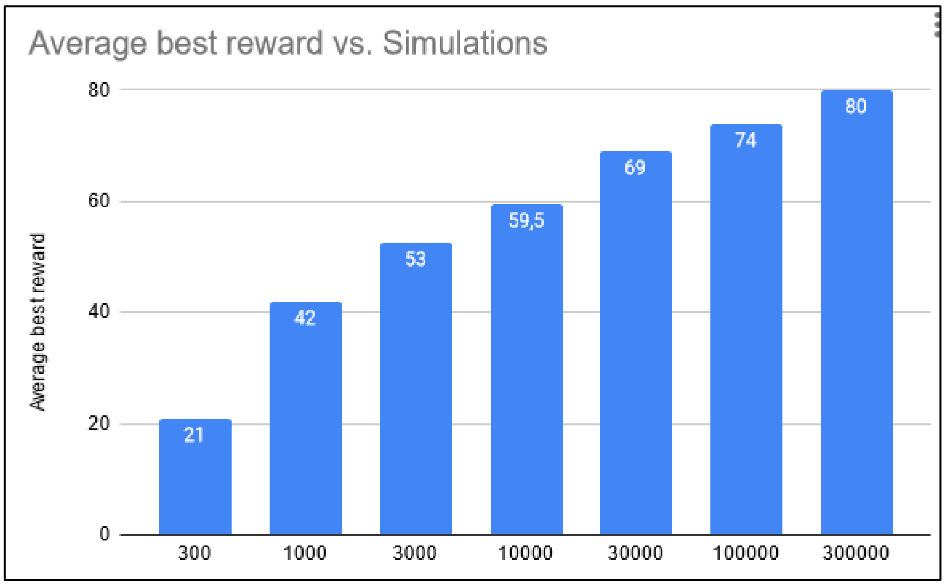


Figure 7 -- Graph of Average best reward after n simulations Monte Carlo-exploration. Removed negatives

find a good path.

It takes exponentially more simulations to get better results. Even with 100000 simulations the Monte Carlo algorithm still can't consistently find the safest path (80 reward).

For Q-learning things look much better for the robot. After just 6 epochs of Q-learning the robot can get a positive reward most of the time. And with more epochs things quickly get even better!

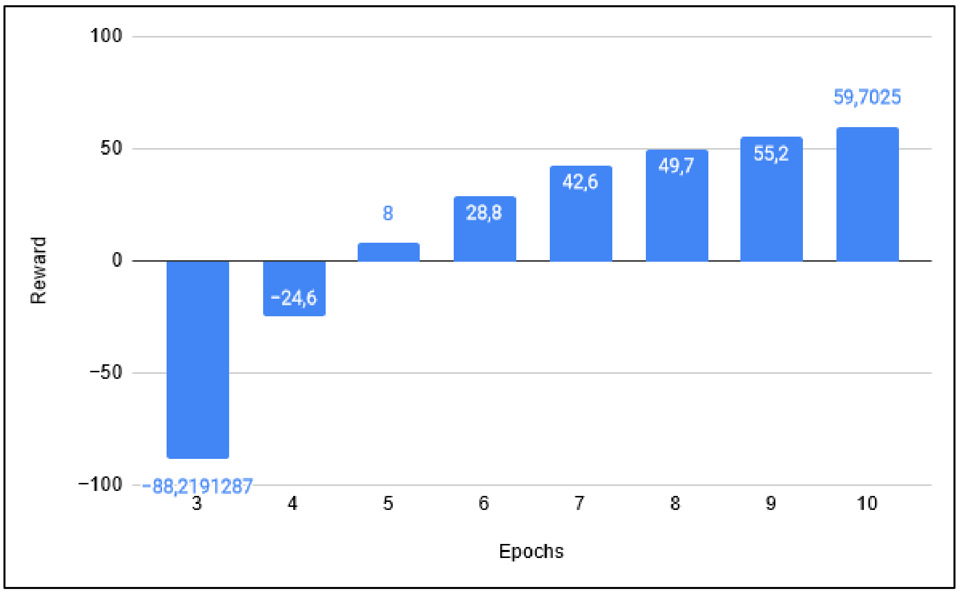
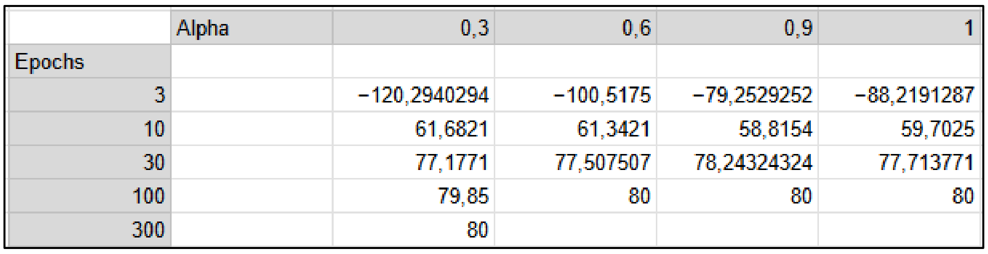


Figure 8 -- Graph of average reward after n epochs Q-learning

After 100 epochs the robot can consistently get the highest reward. Changing the alpha did not seem to make much of a difference. The robot did slightly worse for very low epochs, but it barely makes a difference for higher epoch numbers



	Alpha	0,3	0,6	0,9	1
Epochs					
3		−120,2940294	−100,5175	−79,2529252	−88,2191287
10		61,6821	61,3421	58,8154	59,7025
30		77,1771	77,507507	78,24324324	77,713771
100		79,85	80	80	80
300		80			

Figure 9 -- Table of average reward for various different alpha values and number of epochs

Where changing the alpha did make a difference is in the number of epochs it takes for the Q-matrix to converge.

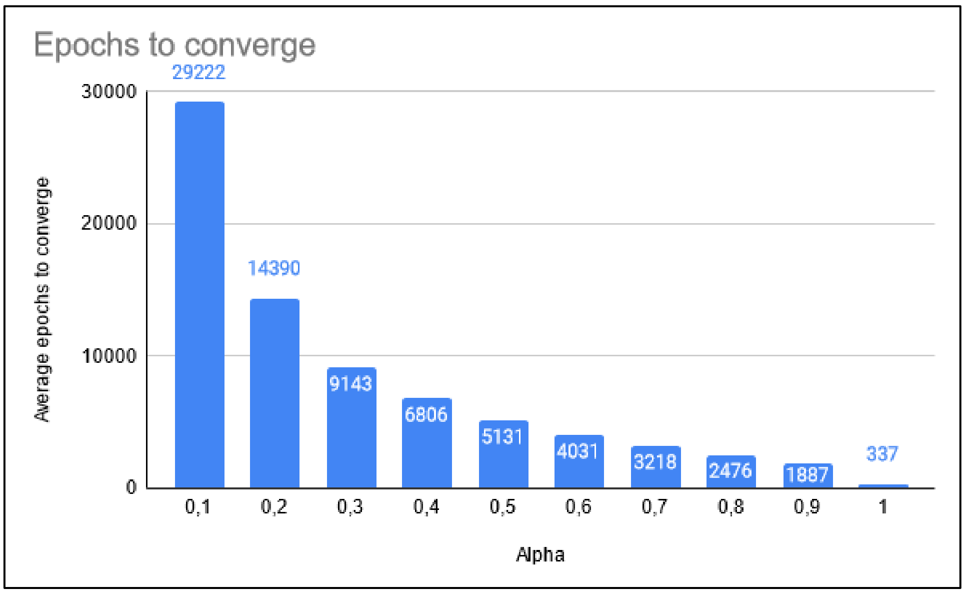


Figure 10 -- Table of number of epochs to converge for different Alpha values

Lower alpha causes the number of epochs required to converge to go up a lot. With just 337 epochs on average for an alpha of 1. And almost 30000 epochs for an alpha of 0,1.

In the map below the Q-matrix values are rendered on the map, with green numbers showing the best action for each state. The safest path from start to goal is also rendered.

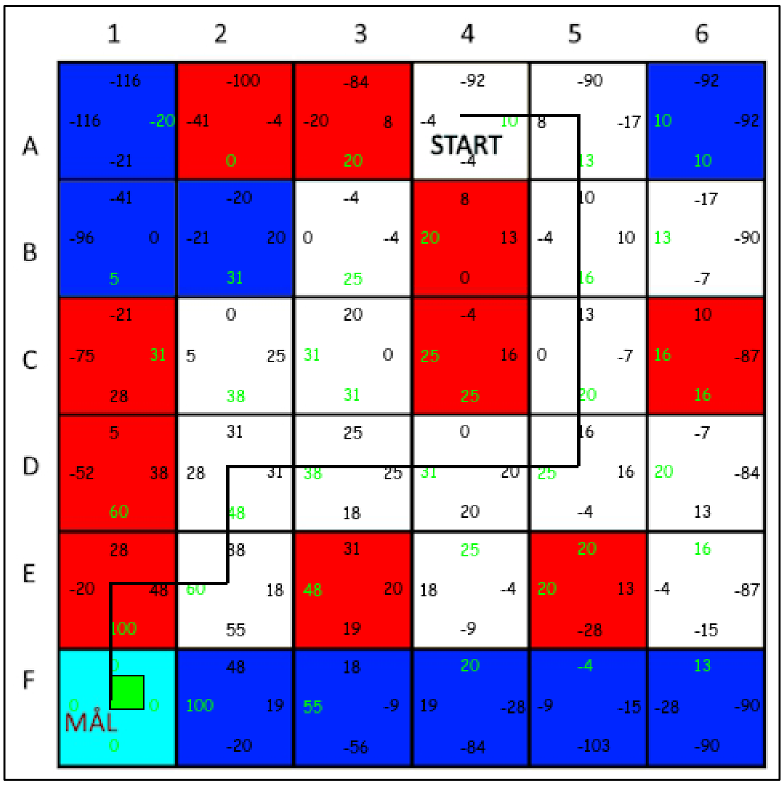


Figure 11 -- Simplified map of terrain with Q-matrix values and safest path rendered

5 Discussion

With the parameters I set, Q-learning worked great, especially when compared to Monte Carlo-exploration. There is only one optimal path, and the map is big enough that just randomly moving has a very low chance of finding the best path. With Q-learning the robot was able to learn the best path. But the parameters must be sensible, changing things like the reward matrix, alpha, gamma, and the policy can break the algorithm. I got lucky that everything worked with my starting parameters, but if I set the reward for reaching the goal higher, or the punishment for going over steep terrain lower, then the robot will take the shorter path over the steep terrain because the reward is so high that taking the time to move around the steep terrain isn't worth it.

Another thing that can break the algorithm is the policy. If the robot only used epsilon-greedy, it would very easily get stuck in an infinite loop. Adding in random moves to avoid getting stuck and to do more exploration is important.

A high alpha was important for the Q-matrix to reach convergence, but for finding a safe path to the end, alpha did not make a big difference. Holding the number of epochs constant, changing the alpha had a very small effect on the best path the robot was able to find. This may be because the robot was still able to learn which path is the best even if the actual values in the Q-matrix are smaller. Also, Epsilon-greedy makes the robot take the best path more often than other paths, so it was still able to learn enough even if the learning rate was low.

6 Conclusion

In this report I compared two algorithms for exploring unknown terrain, Monte Carlo-exploration and Q-learning. Q-learning was a much more effective algorithm for finding the safest path through this terrain. By giving the robot rewards and punishments, it was able to learn how to safely navigate from the start to the goal. While Monte Carlo-exploration had to do thousand of simulations to find a decent path, Q-learning only had to do a few, and it was able to consistently find the best path after just 30 epochs.

I also explored how the alpha value in the Q-learning algorithm changed how many epochs it took for the Q-matrix to converge. An alpha of 1 was optimal, needing only about 350 epochs to converge, while lowering the alpha made the number of epochs go up exponentially, up to 30000 epochs for an alpha of 0,1.

Table of figures

References

[1] g1_qlearning, Github. https://github.com/uit-dte-2602/g1_qlearning/tree/main?tab=readme-ov-file

[2] Q-learning function. Wikipedia contributors, "Q-learning," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=1237581016 (accessed October 13, 2024).