# Q-Learning and SARSA Wall Following Robot

Sahan Prasad Podduturi Reddy

***Abstract*— The motivation behind designing a wall-following Triton LiDAR robot is to address the need for accurately mapping environments in which the robot is placed. This is an important challenge because mastering this task is essential for performing almost all activities in the field of robotics. Real-world applications where this is useful is in the field of autonomous navigation. Perfecting wall following will contribute to safety in autonomous cars and enhance user experience. The main problem that is being addressed by this project is the accurate mapping of indoor environments by the robot and furthermore, intelligent utilization of the data received to navigate through the environment seamlessly without additional user interference. This is a massive contribution to the field of self-driving cars. The main approach we discuss to tackle this problem is Reinforcement Learning. Specifically, we implement Q-Learning which is used to learn a Q-function that can be used to learn the overall value of taking an action in a particular based on a reward system. Eventually, Q-learning outputs a Q-table which contains values that are used to determine which actions to take in a particular state in order to maximize overall reward in the system. We implement two separate algorithms, the Q-Learning and the SARSA algorithms which are able to make the robot efficiently traverse the map following the walls.**

## I. INTRODUCTION

The goal behind the robot wall-following task was to place a robot within a virtual environment in efforts to make the robot learn to follow walls within the environment only using the sensors that the robot is equipped with. Based on the readings that our robot observes, it puts itself into one of various states and performs actions which either keep the robot in the same state or transition the robot to a different state altogether. There is a lot of robot decision making and planning involved because the actions are performed with uncertainty about the outcome of the actions and what states the robot will transition to next in the environment. The approach we followed to teach the robot to navigating through this unfamiliar terrain was reinforcement learning. The robot uses the sensors to interact with the environment and then improves its behaviors based on a reward function associated with not only the state, but also the action that robot takes in that state. So, every action we take produces a reward and the robot tries to take actions in order to maximize reward. In our model, the robot takes actions to maximize cumulative long-term reward instead of greedily opting for immediate gratification. We follow a Markov Decision Process (MDP) in order to define the interactions that our robot will have in the environment based on the actions it takes. Therefore, the history of the robot's past states is not of importance to us. The current state and the information it contains is sufficient for us to take actions to transition to the next state. The robot we have chosen for this problem is the Triton LiDAR robot which is equipped with a 360-degree laser which scan all areas around our robot calculating distances in all directions to the nearest obstacle that we encounter in our environment. The policy function that we define for our RL problem assigns actions to state that are defined by using these laser readings. Finally, the value function of our RL model must be used to predict overall future rewards from a particular state and is used to evaluate how good/bad a state is. This helps with maximizing long-term future rewards. We explore two different algorithms in order to estimate this value function.

In our first approach, we use Q-Learning in our effort to estimate the value function in our problem. The Q-function is recursive in nature and updates values in Q-table which collects information regarding rewards based on taking different actions in different states. Thus, it automatically updates values as the robot moves in the environment. The optimal Q-value is found by this algorithm without any dependency on the policy being followed. In the end, once the Q function is learnt by the robot, we can use the outputted Q-table and information about the robot' current position in order to take actions which have the most Q-value associated with them for that particular state in order to transition to the next state. This way, our robot learns to avoid bad moves such as wall collisions, going too far away from the wall or staying too close to the wall based on our state definition and problem formulation.

$$\text{Q(S,A)} \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$$

The other algorithm we try is the SARSA algorithm. The major difference between this policy and the previous one is that it is as on-policy TD learning algorithm. While Q-Learning does not depend on the current policy to update its values on the next step, SARSA evaluates the current policy which we have defined and improves it to select actions.

$$\text{Q(S,A)} \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$$

For our project, we have implemented both the algorithms to teach our robot how to follow straight walls using RL.

## II. DESIGN

### A. LiDAR Zones

For our robot design, we have split the Triton LiDAR scanner's 360° view into multiple zones which we can utilize in order to define states for our robot. The initial idea was to split the field of view into 8 octants, each covering an angle of 45°. This would give our robot a complete god-view of the entire map. Although this is a powerful technique, we

quickly realized that training our model using these zones to create states will be extremely time-consuming. The reason is that we would have to go through tens of thousands of episodes for our robot to converge on the algorithm and find the optimal path for all scenarios it encounters on the map. With this in mind, and considering the requirements, we decided to tweak the zones in order to count for only the right and front sides of the robot. The reason this will work for us is because we can make the robot a right-facing wall-following robot. This way, the robot only considers the nearest obstacles on the right and front and uses them in order to put itself in one of various states that are defined for this problem. After much trial and error, we ended up with the following zones for our model:

[210:240], [250:290], [310:340], [355:360, 0:15]

Going in the counter-clockwise direction, we cover a lot of the area starting from 210° all the way up to 15° which is the front view of the robot. Thus, we get information from the top-right quadrant and also some from the top-left and bottom-right quadrants.

### B. States

With the information that we get from the angles specified above from the LiDAR scanner, we can now define states for our model. We assign different states to our robot based on distance from the nearest obstacle. We had to first decide on a distance that we wanted the robot to maintain from the wall when following it, this would be the optimal state for the robot while it is trying to follow a wall. Then we defined different states based on distance relative to this optimal state. We ended up with the following states for our robot

1) Very Far: distance $> 0.8$
2) Far: $0.7 <$ distance $\leq 0.8$
3)$Optimal : 0.6 \leq distance \leq 0.7$
4)$Close : 0.5 \leq distance$¡0.6
5)$VeryClose : 0.3 \leq distance$¡0.5
6)$CriticallyClose : 0.11 \leq distance$¡0.3
7)$Collision : distance$¡0.11

We wanted the robot to maintain and distance of 0.65m from the wall when following it. So in this case, Optimal is the optimal state for our robot. We defined two states for when the robot goes far away from the wall and 3 states for when the robot goes too close to the wall. Then, there is a separate state defined for when the robot collides with the wall. We defined this state so that we know when to reset the robot again to a random position on the map and start a new learning episode. Keep in mind that these states are only defined for one of the four angle zones of our LiDAR scanner. Our robot checks the minimum distance from obstacles in all 4 zones and then picks the angle zone with the minimum distance to a wall. Then it assigns a state based on the 7 state definitions we have defined above. Thus, in total there will be **7 x 4 = 28** states for our robot to transition between.

### C. Actions

We went for a simple list of actions that our robot can choose from

Front: Move forward(v = 0.3m/s)
Left: Move left(v = 0.3 m/s, $\pi$ = 0.3 rad/s
Right: Move right(v = 0.3 m/s, $\pi$ = -0.3 rad/s

Since we have a robot that only moves in a single direction, these actions are sufficient for our wall-following problem. At every state, the robot chooses one of the three actions and transitions to different states based on our state definition.

### D. Rewards

To teach the robot what actions are best to maintain optimal distance from the wall while following it, we have defined a reward function which condemns the robot for being in bad states for too long. We decide to go with a reward of zero for the robot when it is in the optimal state and negative rewards for the robot in all other states. The negative rewards get worse as the robot goes further away from the optimal state. Thus, when the robot takes actions and transitions to a worse state, it gets punished more. In this way, the robot learns what actions are optimal to take in a given state. Our reward function is as follows

1) Very Far: -25
2) Far: -10
3) Optimal: 0
4) Close: -10
5) Very Close: -20
6) Critically Close: -30
7) Collision: -100

As you can see the robot is rewarded for being in the optimal state as long as possible. As the robot goes farther away, the negative reward gets worse. When the robot collides with the wall, the negative reward skyrockets. This makes the robot avoid moves that cause wall collisions in close states.

### E. Q-table

We define a Q-table which is used to capture information about the reward accumulated by taking different actions in different states by the robot. For every state, the reward is captured based on which state the robot transitions to when taking a particular action in that state. Since we have 3 actions to choose from, each state will have Q-values corresponding to the 3 possible actions to choose from. Finally, since we have 28 different states, we will have a total of 28 x 3 = 84 entries in our Q-table.

During the training phase, we initialize our robot with all Q-values set to zero and then we let the robot free in the environment to learn based on our defined states, actions and rewards. We let the robot run for several thousands of episodes by spawning it at random locations on the map and letting it learn to get to the Optimal state and follow the wall.

### F. Algorithm Hyperparameters

We make the robot choose random actions at the start of the training phase in order to learn which moves are good/bad in a particular state. We use $\epsilon$ as the deciding factor on when the robot should stop taking random actions to learn and start exploiting what it has learnt so far from the training phase. We start at $\epsilon = 0.9$ and use a decay rate d = 0.95 and gradually decrease $\epsilon$ per episode using this decay rate until $\epsilon$ becomes very small. After this, if the algorithm does not converge, we reset $\epsilon$ to 0.9 and continue training the model. $\alpha$ is the learning rate of the model and it determines how much the previous readings matter to us in the training process. We set $\alpha = 0.3$ for our model to ensure that if our robot accidentally takes bad moves in a particular state, it does not completely override the previous Q-values learnt by the robot during the training process. $\gamma$ is the discount factor. We set $\gamma = 0.8$ to make the model prioritize future rewards more than immediate rewards so that the robot doesn't select greedy actions and get punished for being in bad states in the future.

## III. EXPERIMENTS

### A. Model Configurations

There were several different configurations that we played around with before coming up with the desirable settings for the states, actions, rewards and the hyperparameters for both our Q-Learning and SARSA algorithms. We initially experimented with 8 different LiDAR zones covering the entire front 180° of the robot and 30° extra on both the left and right sides. Each zone was 30° in width.

[0:30], [35:60], [65:90], [95:120], [240:270], [275:300], [305:330], [330:355]

We noticed that since our goal was to make the robot right-facing, the left zones were not very useful and they were some zones at the far left corner were rarely getting updated in the Q-table implying that the robot was not using these zones at all. So we tweaked the zones definition to reduce the zones. This also proved better for training as there are fewer states and the algorithm converges faster.

[0:30], [35:60], [240:270], [275:300], [305:330], [330:355]

Majority of the angles which were closest to the wall came from the same zones and thus the same zone was getting updated for multiple angles of the LiDAR robot. Notably, we wanted a single large front sensor instead of two zones from which the robot kept switching back and forth.

[15:30], [0:10, 350:360], [310:340], [280:300], [255:275], [220:250]

The above configuration worked well for us initially. But, we were unable to make the robot turn left 90° at an L-shaped wall. After more tweaks, we ended up with the following states which were perfectly suited for both our Q-Learning and SARSA algorithms.

[0:15, 355:360], [310:340], [250:290], [210:240]

These final angle zones with our current state definitions helped the model converge for both algorithms. We also had to play around with the reward function to reach optimal configurations. We initially had a reward of -100 for the 'Very Far' state and -50 for the 'Far' State. Although this helped prevent the robot from wandering far away from the wall, the robot was more prone to colliding with the wall this way and it was leading to lots of episodes with a smaller number of total time steps completed.

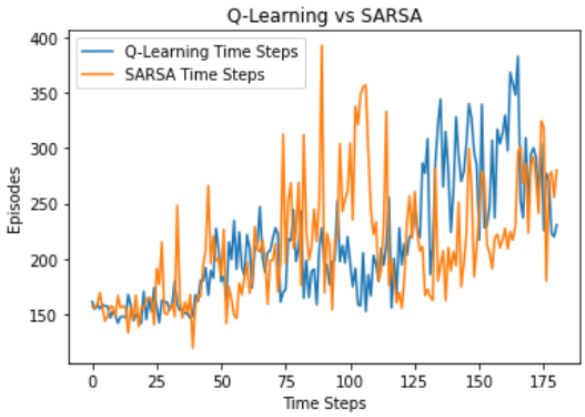
### B. Number of Episodes

Both algorithms took a long time to converge. A possible reason for this was that there were a lot of states for the robot to go through during the training process and it naturally takes time to accurately define rewards for taking all possible actions in all possible states. Overall, the **Q-Learning algorithm took around 8,000 episodes to converge**. The **SARSA algorithm took around 7,500 episodes to converge** for my model. So, overall there wasn't a big difference in terms of computational costs to use either of the algorithms to train my robot for the wall following problem based on the states and the policy function defined.

### C. Average Time Steps

In our algorithm design, the $\epsilon$ value keeps decreasing according to our decay rate. So after a certain point, the$\epsilon$ value falls very low and the robot stops learning altogether and starts exploiting. In order to make the robot learn more, we have to reset the $\epsilon$ value and continue running the script. According to our script, the $\epsilon$ value falls very low after around **180 episodes** and we reset the $\epsilon$ value to continue the learning process.

Our algorithms are currently programmed to terminate if the total reward accumulated in an episode is below -4000. So, to determine whether our robot has lerant, we check the average time steps that the robot travels before termination based on the episode number. Since $\epsilon$ values reduce with increase in episode number, we expect more exploitation of data at higher $\epsilon$ translating to more distance traveled by the robot prior to termination.

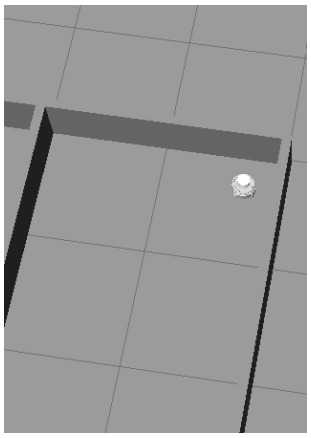



According to the graph above, we notice that the robot travels greater distances as episod counts increase. We also

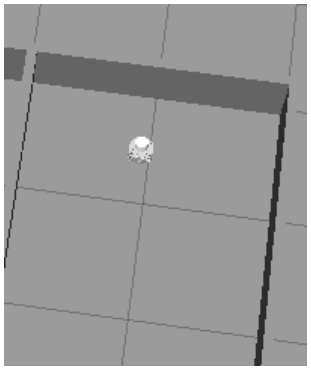
notice that although the SARSA algorithm leads to faster initial progress, the Q-Learning algorithm picks up and both of them converge at around the same points. So in our case, both the algorithms were equally effective in training the robot for wall following in this environment.

### *D. Complex Scenarios*

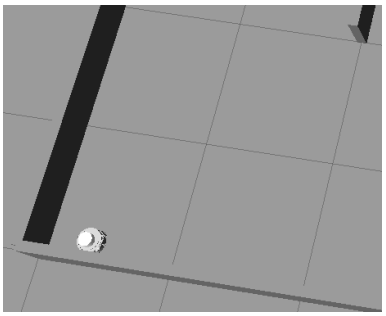
Two scenarios that the robot had trouble learning were the U-Turn scenario in the map and turning left 90° at an L-shaped corner. We have snapshot showing how after the training process, the robot finally learns to properly navigate through the environment.



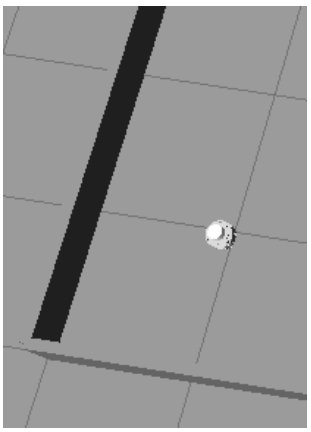
The picture above shows our U-Turn scenarios in the earlier phases of the training process. Initially, the right sensors used to push to robot into the wall causing collisions.



The final result after training shows the robot able to properly complete a 180° turn. This is because the robot properly learnt to make a left turn in the proper zone in which the minimum distance is in front of the robot.



Similarly, the left turn at the L-shaped wall problem was also caused by the robot's right sensors pushing the robot into the nearest wall on the right. This was also solved after many episodes of training.



## IV. CONCLUSIONS

In the end, we notice that both the Q-Learning and SARSA algorithms are equally efficient in training a wall-following robot. Although the SARSA algorithm produces higher time steps quickly in our graph, this is probably due to the random steps that the robot takes earlier on due to having s higher $\epsilon$ value. This is because the time steps drop in successive episodes and in the end, both algorithms produce almost the same average time steps after 180 episodes. Overall, solving this problem is a very creative solution to the problem of accurate and efficient mapping of indoor environments. This study also shows how Reinforcement Learning can be leveraged to solve problems that can further enable the performing of other complex Robotics tasks. Mapping is also crucial in the field of Virtual Reality and the perfection of mapping can expand our VR experience significantly. This along with the possibility of massively enhanced safety features from the implementation of perfect mapping solutions in real world scenarios such as in the field of self-driving cars. The accuracy of the measurements also help in shoreline mapping and hydrographic surveying which are some of the many areas in which LiDAR is in use today.