# Trajectory Optimization for Autonomous Parking Lot Cleaning using Q-Learning

Alban Broze
*Institute for Computational and Mathematical Engineering*
*Stanford University*
Stanford, United States of America
abroze@stanford.edu

Thomas Sounack
*Department of Mechanical Engineering*
*Stanford University*
Stanford, United States of America
tsounack@stanford.edu

***Abstract*—In this paper, we present an algorithmic solution for the optimization problem of sweeping a parking lot with a self-driving truck. The problem is approached by modeling it as a variant of the Traveling Salesman Problem (TSP), where the parking lot is discretized into a set of stops, and the objective is to find the optimal path that visits all the stops without revisiting the initial one. The solution is based on Q-Learning, a model-free method, due to the large state space. The state space and action space of the problem are described, and the reward system is presented. We also discuss the construction of the environment (the parking lot and the truck) and the addition of uncertainty to the model. Finally, we demonstrate the feasibility of our approach by showing simulations of the truck's trajectory on random sample grids.**

***Index Terms*—trajectory optimization, Q-Learning, TSP, reinforcement learning, uncertainty, algorithm**

## I. Introduction

The project consists of a partnership with Hermes Robotics, a startup designing kits to retrofit trucks into self-driving vehicles. The trucks are then used to autonomously clean parking lots. The objective of this project is to provide an algorithm that can compute an optimal trajectory for the truck to sweep an entire given parking lot by considering its boundaries and obstacles.

## II. Chosen Approach

### A. Representation of the Problem

To solve this task, we took our inspiration from the Traveling Salesman Problem (TSP). The TSP is a famous combinatorial optimization problem that attempts to find a solution to the question: "given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? [1]" The TSP was mathematically formulated in the 19th century by the Irish mathematician William Rowan Hamilton and by the British mathematician Thomas Kirkman [1].

By discretizing the given parking lot to a set of well-chosen stops (cities), we can assume that visiting all the stops in the set comes down to exploring the entire parking lot. The way the stops are chosen and the environment is created is crucial and is detailed in section II-B. The only difference in our implementation compared with the actual TSP is that, in our problem formulation, it is not required to come back to the initial state after visiting all the others. Also, note that the truck cleans the parking while driving.

The set of nodes (stops) constitutes the state space of our MDP. The action space is also characterized by the set of nodes but is dynamic. Initially, all the nodes except the one corresponding to the initial position can be reached. Then, as exploration progresses, the nodes that have already been visited are kept in memory and can't be reached anymore. This ensures that an optimal exploration trajectory can be found. This yields $\mathcal{A}_n = \mathcal{S} \backslash \mathcal{V}_n$, where $\mathcal{A}_n$ is the action space after having visited $n$ nodes, $\mathcal{V}_n$ is the set of nodes that have been visited after visiting $n$ nodes, and $\mathcal{S}$ is the state space containing all the nodes in the parking lot. The rewards system will be covered in section II-D.

Since the grid's size depends on the size of the parking lot and the vehicle (see section II-B), the state space can be very large. As a result, exact solution methods may potentially require an infeasible amount of computation. As the objective is to compute a path prior to the cleaning of the parking with a known parking geometry, it was chosen to implement the model-free method Q-Learning to solve this problem. The implementation will be covered more in detail in section II-E.

### B. Constructing the Environment

In order to construct the environment (representing the parking lot and the truck), we generate a grid world where each node has to be visited and cleaned by the vehicle. We define the dimension of the grid, the width of the vehicle as well as the coordinates of various obstacles in the grid, and a function creates the nodes according to the following criteria:

1) Initially, the nodes are uniformly distributed over the grid at regular intervals, with a distance corresponding to the width of the vehicle between them to avoid sweeping the same locations multiple times.
2) Then, the boundary nodes are separated from the borders of the grid world with a distance corresponding to the width of the vehicle.

3) Finally, nodes that are too close to obstacles are shifted away from the obstacles by a distance equal to the vehicle's width.

The last two steps mentioned above are requirements imposed by Hermes Robotics to avoid a collision of the truck with obstacles. Fig. 1 shows an example of a grid representing a parking lot.

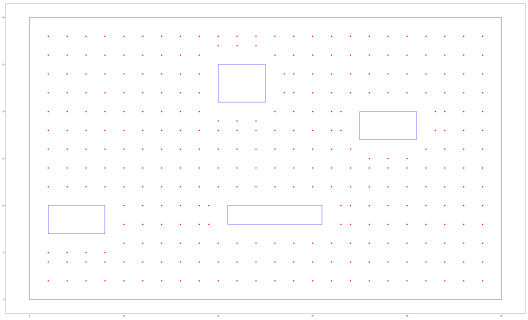

Fig. 1. Example of a grid world built with the parking lot's constraints.

### C. Adding Uncertainty

The initial project given by Hermes Robotics did not contain an uncertainty component. However, we decided to add some uncertainty to the model in order to make it more realistic. As such, Gaussian noise was added to the dimensions of the obstacles. Thereby, there is uncertainty in assessing whether passing from one node to the other results in a collision with an obstacle, as explained in section II-D. In real life, this could correspond to using a LiDAR sensor with Gaussian noise to determine the distance to an obstacle.

### D. Reward System

Since, initially, our action and state spaces are identical (and correspond to the set of nodes describing the parking lot), the action-value function $Q(s, a)$ is a square matrix. We initialize it using the following rule:

$$Q_{i,j} = -\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$

which allows us to initialize the utilities to values that are proportional to the Euclidean distance between nodes. This encourages the truck to travel to closer nodes first.

In order to take the obstacles into account, we heavily penalize a trajectory that requires the truck to go through an obstacle to continue its path. This is translated in a function that detects when the line segment linking two successive nodes of the path intersects with an obstacle. Such behavior can be depicted in Fig. 2.

In reality, having a line segment between two nodes that crosses an obstacle is not a big problem since the truck can simply go from one node to the other by passing around that obstacle. Trajectories to go from point A to point B by avoiding obstacles can be generated by algorithms like RRT or A* [2], but these algorithms were not implemented since they are out of the scope of this project. However, this behavior of crossing an obstacle is penalized in our implementation as it requires the truck to sweep a surface that will be swept anyways.

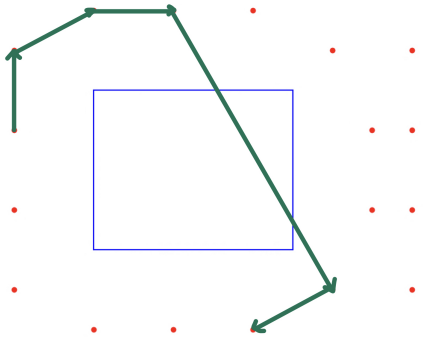

Fig. 2. Example of a trajectory intersecting with an obstacle

### E. Implementing the Algorithm

Using the textbook [3] as well as a website applying reinforcement learning to the TSP [4], we were able to articulate our project around the implementation of our own instance of the problem in Python.

To implement our Q-Learning algorithm, we use an agent with an $\epsilon$-greedy policy and an exponential decay schedule. Several runs (events) are simulated and, for each run, the agent has to visit every single node, i.e. explore the entire parking lot. This ensures that the whole surface is cleaned. The run with the overall best reward accumulation is saved along with its path, which corresponds to the optimal policy for the truck and is the best solution to the problem we are trying to solve.

## III. Analysis and Results

### A. Choosing the Hyperparameters

In order for the Q-Learning algorithm to yield a satisfying solution to our problem, we need to choose its hyperparameters appropriately. The hyperparameters that were considered and evaluated are: the learning rate $\alpha$, the discount factor $\gamma$, the $\epsilon_0$ for the greedy exploration, the exploration decay parameter $\eta$, and the number of episodes to run.

In reinforcement learning applications, it is common to have a constant $\alpha$. This parameter makes sure that the weights of older samples decay exponentially at the rate $(1 - \alpha)$. A low value of $\alpha = 0.05$ was chosen since, even though convergence takes longer, this yields more stable results with fewer fluctuations.

Since we want our algorithm to be forward-looking, i.e. to consider long-term rewards more than short-term rewards,

we opted for a $\gamma$ close to 1, $\gamma = 0.95$ precisely. This choice also leads to slower convergence but it is crucial that our algorithm is not myopic if we want it to solve an instance of the TSP.

Next, $\epsilon$-greedy will be discussed. This method enables our learning agent to explore its environment stochastically. More precisely, $\epsilon$ is a hyperparameter that dictates at which rate we want our agent to explore random actions to take instead of taking the greedy action that would maximize immediate returns. Since we want our agent to explore its environment a lot at the beginning of its learning process, we start with a high value of $\epsilon_0 = 100$, meaning it starts by exploration only. However, we don't want our agent to explore randomly forever. This is why we decided to implement a discount factor $\eta$. We chose a high discount factor of $\eta = 0.99$. After each action taken, $\epsilon$ is updated as follows:

$$\epsilon \leftarrow \epsilon * \eta \tag{1}$$

This way, exploration decays over time as learning progresses. It is important to note that this algorithm chooses random actions every time while $\epsilon \geq 1$. Then, once $\epsilon < 1$, the algorithm starts choosing the greedy action with probability $(1 - \epsilon)$. The number of episodes $N$ needed for $\epsilon < 1$ is given be the following equations:

$$\epsilon_0 * \eta^{N*|\mathcal{S}|} < 1 \tag{2}$$

Solving for $N$ gives:

$$N \geq \frac{-ln(\epsilon_0)}{\ln(\eta) * |\mathcal{S}|} \tag{3}$$

This is because, for each run, $|\mathcal{S}|$ actions are taken.

Finally, we were interested in knowing how many episodes were needed for our algorithm to converge to an optimal solution. For this, we plotted the reward evolution over the number of episodes for environments with 1 to 6 obstacles. This is shown in Fig. 3

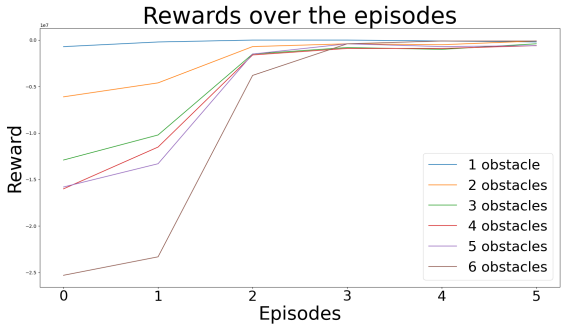


Fig. 3. Evolution of rewards over the number of episodes

It can be seen from Fig. 3 that the rewards for all the obstacle scenarios seem to converge after 3-5 episodes. This is the result of choosing the above-discussed hyperparameters appropriately. Furthermore, having to run only 5 episodes is computationally very efficient. Indeed, running 5 episodes for 6 obstacles takes less than 1 second on a local machine.

Even if only 5 episodes would be required to achieve satisfying results, we decided to run the simulations for 100 episodes. This was possible thanks to our efficient implementation because 100 episodes take only about 10 seconds to be computed for 6 obstacles. Furthermore, simulating the agent learning process for more episodes increases the probability of obtaining an optimal trajectory to sweep the entire parking lot.

The chosen hyperparameters are summarized in Table I.

| Hyperparameters | $\alpha$ | $\gamma$ | $\epsilon_0$ | $\eta$ | N episodes |
|---|---|---|---|---|---|
| Values | 0.05 | 0.95 | 100 | 0.99 | 5-100 |

TABLE I
SUMMARY OF THE CHOSEN HYPERPARAMETERS

In addition, the impact of the number of episodes on the obtained reward for different numbers of obstacles can be retrieved from Table II.

| # of obstacles | Max. reward (5) | Max. reward (100) | % increase |
|---|---|---|---|
| 1 | -747.9 | -675.0 | 9.7 |
| 2 | -100640.5 | -630.8 | 99.4 |
| 3 | -400627.2 | -100629.9 | 74.9 |
| 4 | -600691.3 | -300688.3 | 49.9 |
| 5 | -400636.0 | -100634.7 | 74.9 |
| 6 | -100557.0 | -549.8 | 99.5 |

TABLE II
IMPACT OF THE NUMBER OF EPISODES (5 AND 100) ON THE OBTAINED REWARD FOR DIFFERENT NUMBERS OF OBSTACLES

It can be seen from Table II that, even though Fig. 3 indicates that reward convergence happens after 5 episodes, running more episodes results in better maximal rewards. This can be explained by the fact that the Reward axis in Fig. 3 has a huge range of values. This analysis led us to run 100 episodes to train our agent.

### *B. Analyzing the Policies*

After training our agent on a certain number of simulations, it is able to find the optimal trajectory for the truck to sweep the entire parking lot as fast as possible. In this section, we will show you the resulting policy in the form of a grid world with the same types of elements as described in Fig. 1. The randomly selected starting and ending points for the trajectory are annotated by **START** and **END** on the grid. The resulting trajectories for 2, 4, and 6 obstacles can be seen in Fig. 4, Fig. 5 and Fig. 6, respectively.

It can be seen from Fig. 4, Fig. 5 and Fig. 6 that our algorithm is able to find an optimal trajectory that sweeps the whole surface of the parking lot. Sometimes, intersections with obstacles like in Fig. 2 were observed but, as explained in subsection II-D, this is not a big problem as long as this

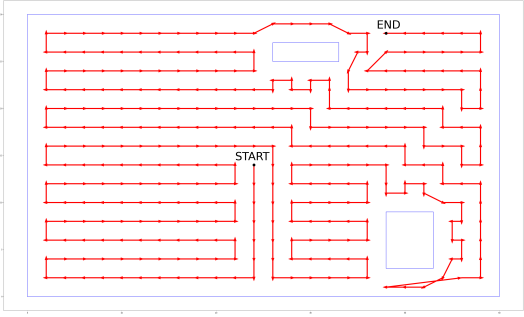


Fig. 4. Optimal Trajectory for 2 Obstacles

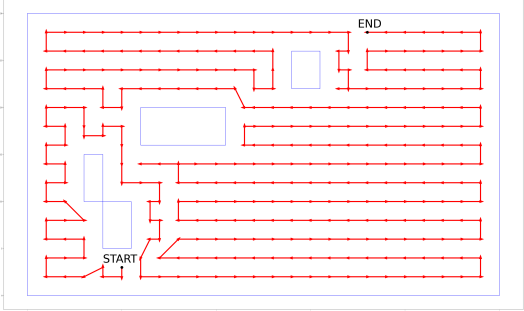


Fig. 5. Optimal Trajectory for 4 Obstacles

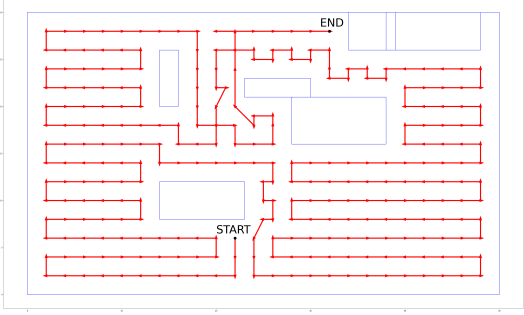


Fig. 6. Optimal Trajectory for 6 Obstacles

behavior is penalized in the reward shaping.

In general, the more obstacles a grid contains, the more likely it is that an intersection might occur. To quantify the expected obstacle intersection rate, an experiment was conducted. For each number of obstacles (ranging from 1 to 6), 10 optimal policies were computed using our algorithm. For each number of obstacles, the rate of obstacle intersection was computed. Eventually, the average rate of obstacle intersection was calculated, which corresponds to the expected obstacle intersection rate. The results of this experiment can be found in Table III.

| Number of obstacles | 1 | 2 | 3 | 4 | 5 | 6 | Average |
|---|---|---|---|---|---|---|---|
| Obstacles intersected (%) | 0 | 15 | 3.3 | 7.5 | 14 | 15 | 9.14 |

TABLE III
AVERAGE INSTANCES OF OBSTACLE-CROSSING IN THE POLICY

It can be concluded from Table III that the expected rate of obstacle intersection is close to 9%. This is a satisfying result because this means that our algorithm is capable of combining obstacle avoidance and exploration for about 91% of the obstacles. For the remaining 9%, the truck simply has to make a small detour to avoid the obstacle, as explained in subsection II-D.

## IV. Conclusion

The objective of this project was to provide an algorithm that can compute an optimal trajectory for the truck to sweep an entire given parking lot by considering its boundaries and obstacles. Our approach as well as our results have been detailed in this report. To conclude, it can be said that our algorithm is able to compute the optimal trajectory by taking the parking lot's boundaries and obstacles into consideration. This follows from the fact that we chose the model-free method Q-Learning and were able to represent our problem as an instance of the Traveling Salesman Problem in a smart way. Furthermore, the robustness of the reward system and the efficient implementation of the algorithm enable the algorithm to run enough simulations to find trajectories that minimize both the number of obstacle intersections and the path length. Finally, by tuning the hyperparameters associated with the problem effectively, optimal trajectories can be found for randomly generated configurations.

The next steps will be to adapt our optimal trajectory to a smoother trajectory that includes the truck's dynamics and to present the results to Hermes Robotics.

## V. Distribution of the Tasks

In this project, collaboration was key. We planned meetings regularly to work on the problem representation and the implementation together. As such, almost all of the work of this project has been equally distributed.

## REFERENCES


[1] Contributors, "Travelling salesman problem," Available at https://en.wikipedia.org/wiki/Travelling_salesman_problem, (NA).

[2] A. Patel, "Introduction to A*," Available at http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html, (2023/02/20).

[3] K. H. W. Mykel J. Kochenderfer, Tim A. Wheeler, Algorithms for decision making, 5th ed. Cambridge MA: MIT Press, 2022.

[4] T. A. D. Costa, "Solving the traveling salesman problem with reinforcement learning," Available at https://ekimetrics.github.io/blog/2021/11/03/tsp/?fbclid=IwAR2nQjoZFURlvJO1DivclBndj5dFcTLJ9bN3sID_FxXqrOo5S1bCZO2OB-w, (2021/11/03).