MSc Control and Robotics
Academic year 2023 - 2024

AUVE Lab 2

Raffaele Pumpo
Emanuele Buzzurro
Under the supervision of Charlotte Beaune

Contents

1 Introduction

In this lab, we explore the realm of robot path planning in a real-world environment using OpenStreetMap (OSM) data. The lab is divided into three main parts, each addressing crucial aspects of autonomous navigation.

1.1 Part 1: OSM Map Loading and Visualization

The initial part of the lab focuses on exploring the capabilities of OpenStreetMap data with the aid of the osmnx library. OpenStreetMap is a collaborative mapping project that empowers individuals worldwide to contribute, edit, and share geospatial data. Unlike traditional mapping services, OSM is an open-source platform where volunteers, communities, and organizations collaboratively create a detailed and up-to-date map of the world's geographical features.

By leveraging this versatile tool, we gain the capability to load OSM maps for specified locations and visualize the associated graphs, nodes, and edges. OSM provides a wealth of information about road networks, pedestrian paths, and various geographic features.

As we delve into the world of cartography, we'll not only visualize the OSM graph for a designated location but also employ the $ox.plot\_graph\_route()$ function to plot paths on the graph. The resulting paths represent potential trajectories for a robot navigating the intricate web of roads and pathways in the given locale. To further enhance our understanding, we'll convert these global coordinates into local coordinates, paving the way for seamless integration with subsequent components of our robotic system.

1.2 Part 2: Global Planning with A^* Algorithm

In this part, the focus shifts to the essentials of global path planning. We begin by loading a pre-existing road network graph, Town 01 Road Network, and establishing the positions of key actors, notably the ego vehicle. The objective is straightforward: find a route from the current position of the ego vehicle to a predefined target.

To accomplish this, we introduce the A* algorithm, a robust method for traversing graphs. Acting as a virtual navigator, A* efficiently explores the nodes and edges of the road network, identifying the optimal path. Integral to A*'s decision-making process is a heuristic function, a guiding principle that gauges the distance between points, tuning the exploration for precision.

This task paves the way for a more in-depth understanding of algorithmic navigation, where the synergy of graph theory, heuristic assessment, and digital maps converges to chart a seamless course from origin to destination.

1.3 Part 3: Local planning with Dynamic Window Approach

Transitioning into the third part of the lab, we shift our focus to the Dynamic Window Approach (DWA), a fundamental algorithm for robot motion planning. DWA dynamically computes feasible trajectories by considering the robot's current state, environmental obstacles, and desired target positions. The algorithm explores the robot's "dynamic window," comprised of achievable linear and angular velocities, to determine the optimal trajectory.

Through the lens of DWA, we'll simulate the robot's motion and dynamically adjust its trajectory to navigate around obstacles while striving to reach specified waypoints. The real-time visualization of the robot's path serves as a tangible demonstration of the algorithm's efficacy in handling the complexities of a dynamic environment.

Throughout this lab, we leverage powerful Python libraries such as osmnx and matplotlib to seamlessly integrate map data, visualize graphs, and animate robot trajectories. By combining theoretical concepts with practical implementations, this lab aims to provide a hands-on experience in robot path planning, a critical aspect of autonomous systems in diverse real-world scenarios.

2 Part 1 - Load and visualize OSM Maps

2.1 Task 1

The goal of this first task was about choosing a desired position coordinates, that we were able to find by selecting a location of interest using an OpenStreetMap map, and then we had to visualizes the OSM graph, nodes, and edges for the given location.

To visualize it, we had to use the *osmnx.plot* module, and according to its documentation, the right function is the following:

$$ox.plot.plot\_graph(G, ax = None, figsize = (8, 8), node\_color =' w', node\_size = 15)$$

In this report have been omitted all the input related to the graphical part, while the highlighted elements, are respectively:

- **G**: input graph
- **ax**: if not None, plot on this preexisting axis
- **figsize**: (tuple) – if ax is None, create new figure with size (width, height)
- **node_size**: (int) – size of the nodes: if 0, then skip plotting the nodes

The resulting figure is the following:

Figure 1: Graph Plot

2.2 Task 2

In the second task of the first part, the goal was about plotting, on the graph previously obtained, the path from origin to destination, using *nx* (network) library. This has been done using the following command line:

$route = nx.shortest\_path(G, origin\_node, destination\_node, weight =' length')$

Where:

- **G**: NetworkX graph, it is the graph to be searched.
- **source**: Starting node for path. If not specified, compute shortest paths for each possible starting node.
- **target**: Ending node for path. If not specified, compute shortest paths to all possible nodes.
- **weight**: Weight of the edge attribute used as the edge weight/distance/cost in shortest path algorithms

After that, we plot it using the function *ox.plot_graph_route*() as following:

$osmnx.plot.plot\_graph\_route(G, route)$

Which takes as input, always omitting visual elements, the input graph G and the route as a list of node IDs, giving us the following figure:

Figure 2: Shortest Path Plot

2.3 Task 3

In this last task of the first step, the goal was to convert the coordinates from $GeoDataFrame$ into $East-North-UpCoordinates$. This system uses the Cartesian coordinates (xEast,yNorth,zUp) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (lat0,lon0,h0) as shown in the figure below:

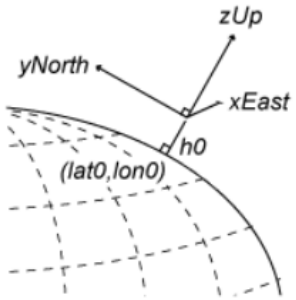

Figure 3: ENU Coordinates System

To successfully complete this task, we used the $gdfs\_to\_local$ function in the following way:

$wpts = gdfs\_to\_local(nodes\_proj, route)$

Where it takes as input:

- **nodes_proj**: (gpd.GeoDataFrame) - a GeoDataFrame of nodes
- **route**: a list of nodes' ids

And it gives us, as output:

- **waypoints_ENU**: (np.array): an array of shape (len(path), 3) containing the coordinates of the nodes in the local frame

In the end, using the plot function we obtained our desired path, converted in local coordinates, as shown in the image below:

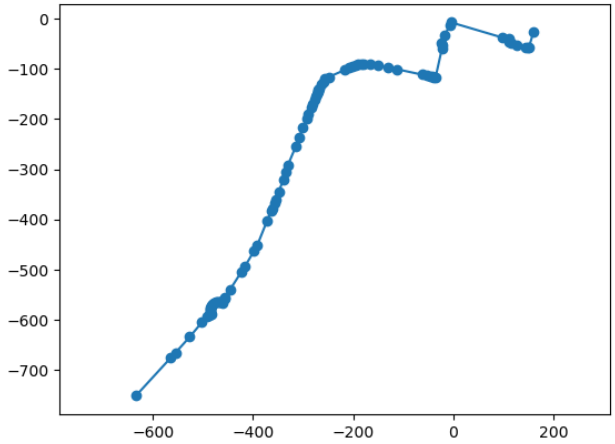

Figure 4: Shortest Path in local coordinates

3 Part 2 - Implement a Global Planner

The first goal of this second part, is about loading and drawing a pre-existing road network graph (Town 01 Road Network) and actor positions.
To do that, we used networkx and matplotlib functions, as following:

$plt.figure(figsize = (10, 10))$ # set figure size

$nx.draw(G, pos = node\_positions, node\_size = 1, node\_color =' k')$ # draw the graph

Where the first one is just used to select the size of the plot, while the second one, draw the graph as a simple representation with no node labels or edge labels and using the full Matplotlib figure area and no axis labels by default, and it takes as input G (networkx graph) and the Node positions.

The resulting plot is shown in the figure below:

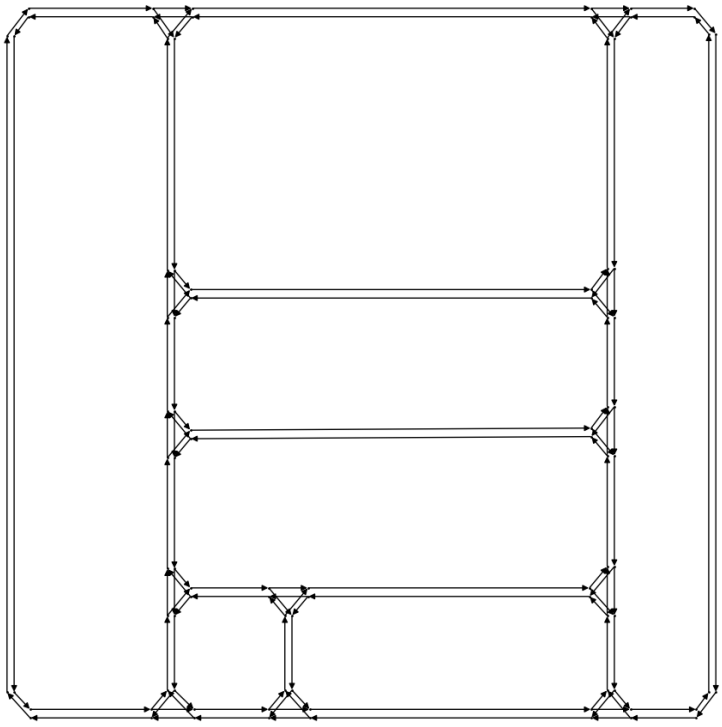

Figure 5: Town 01 Road Network Graph

After that, we had to implement a custom A^* algorithm, which find the shortest path between two nodes in a graph, taking as input:

- **graph**: networkx graph
- **start**: (int) - starting node
- **end**: (int) - ending node

And giving as output:

- **path**: list of nodes in the path

The main idea behind it, is first to initialize two cost functions:

-g_score(Actual Cost from Start to Current Node): represents the actual cost incurred from the start node to the current node along the path under consideration. It is a cumulative measure of the distances traversed and is continuously updated as the algorithm explores the graph. In simpler terms, it keeps track of how far we have traveled from the starting point to reach the current node.

-f_score (Estimated Total Cost from Current Node to Goal): is an estimate of the total cost expected from the current node to the goal node. It is a combination of the actual cost (g_score) and a heuristic estimate of the remaining cost. The heuristic function provides an informed guess about how much farther we need to go to reach the goal.

In this implementation, the chosen the **Heuristic Function** is the Euclidean distance between two nodes. Which is a straightforward measure of the straight-line distance between two points in a two-dimensional space. In this case, it estimates the "as-the-crow-flies" distance between the current node and the goal node. Mathematically, the Euclidean distance between two nodes, node and goal, is calculated as follows:

$$distance = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$$

Where x_1,y_1 indicates the cartesian position of the current node, while x_2,y_2 indicates the cartesian position of the global node.

After having specified that, the algorithm iterates as long as the open_set is not empty, which represents the set of nodes that still need to be evaluated; then it selects the node with the minimum f_score from the open_set as the current node and if the current node is the goal node (end), the algorithm has found a path. It reconstructs the path from the goal node to the start node by backtracking through the came_from dictionary.

Starting from the goal node, the algorithm backtracks using the came_from dictionary. It appends each node to the path list until it reaches the start node. The path list is then reversed to obtain the correct order. The final path is returned once it has been successfully reconstructed.

In the end, if the current node is not the goal, it is removed from the open_set, and its neighbors are explored. For each neighbor, a tentative g_score is calculated by summing the actual cost from the current node to the neighbor. If this tentative g_score is lower than the current g_score for the neighbor, the came_from, g_score, and f_score are updated. The neighbor is then added to the open_set if it's not already present.

In order to apply this algorithm in our "Town 01 Road Network " Graph, we need to find starting and ending node, and it can be done as explained below:

- **starting node**: node with minimum Euclidean distance between node itself and vehicle
- **ending node**: freely choosen

And at this point we have just drawn the graph and the path on top of it, obtaining the following image:

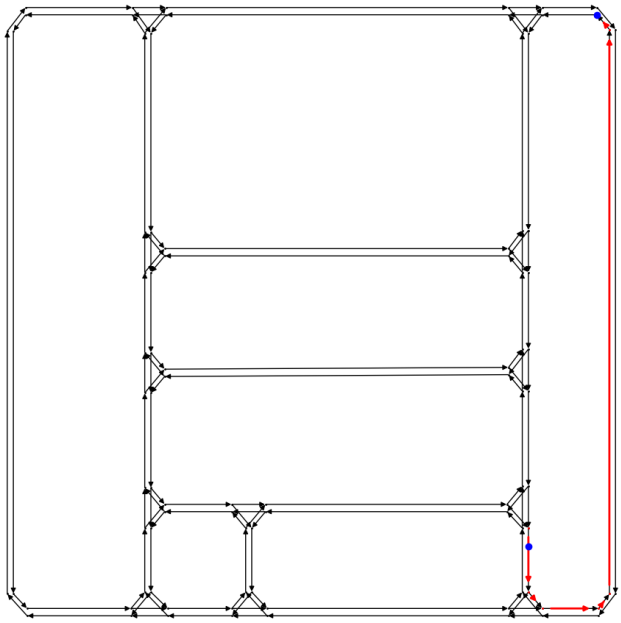

Figure 6: Path given by A^* algorithm

By looking at the nodes highligthed in red, and at the starting position of the car in blue (which have been added additionally by us), we can observe that the initial node do not coincide exactly with the initial vehicle position, and that is because we just look for the nearest one, in order to properly set the best path.

4 Part 3 - Implement a Local Planner

4.1 DWA

In this task, we had to implement the Dynamic Window Approach (DWA) for local planning. The algorithm explores various combinations of linear and angular velocities within specified ranges to generate trajectories for the robot. For each combination, a short-term trajectory is simulated, considering the robot's kinematic model. The cost of each trajectory is then calculated, taking into account factors such as minimizing the distance to the target, maintaining a desired speed, and penalizing collisions with obstacles.

The algorithm iterates through the velocity combinations, and for each, it evaluates the trajectory's cost. The trajectory with the minimum cost, indicating the most optimal path, is selected. The resulting best trajectory and the corresponding linear and angular velocities are returned. In summary, the DWA dynamically adjusts the robot's velocity based on the surrounding environment, enabling it to navigate efficiently while considering both its current state and the target destination.

To implement it correctly we needed, firstly, to define various parameters such as maximum velocity, acceleration, and rotation. Then we defined the functions to calculate a cost for a given trajectory:

- **Distance Cost**: Euclidean distance between the final position of the trajectory and the target.
- **Speed Cost**: Accumulated angular velocity changes over time in the trajectory.
- **Collision Penalty**: enalizes collisions with obstacles. If the distance between the robot's position and any obstacle is less than 10.0, it accumulates a penalty.

And last but not least we also needed a function able to update the state: *evolve* which take as Inputs:

- **current_state**: Current state of the robot (position and orientation).
- **v**: Linear velocity.
- **w**: Angular velocity.

It basically updates the robot's position and orientation based on the linear and angular velocities using the kinematic model and returns the new state as a numpy array.

To test this algorithm, we used an example, that uses the Dynamic Window Approach (DWA) to plan the motion of a robot through a simulated environment. The robot's initial state is defined by its position (x, y) and orientation (theta). The simulation runs for 100 iterations, during which the robot dynamically plans its trajectory based on the DWA algorithm.

The loop checks whether the robot is close to the next waypoint and updates the waypoint accordingly. The DWA function is then called to compute a trajectory, considering the robot's current state, the target waypoint, and the absence of obstacles initially (represented by an empty list).

As the simulation progresses, the resulting trajectory is plotted in red dots, illustrating the robot's positions along the simulated path. The trajectory is plotted in blue. The trajectory lines connect consecutive red dots, which represent the possible trajectories, providing a visual representation of the robot's path through the environment, obtaining the following plot:

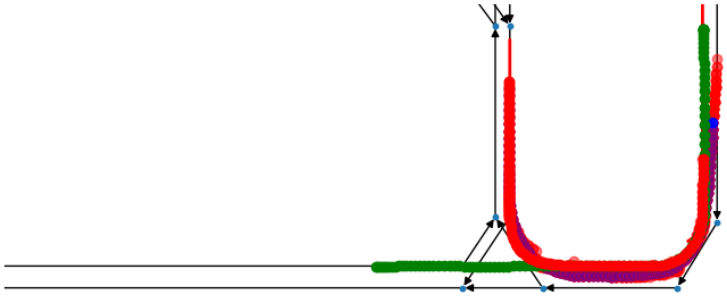

Figure 7: Path given by DWA Algorithm with dist cost weight 100x the speed cost weight

Also the plot is adjusted to focus on a specific region for better visualization, with the x-axis limited to the range (200, 400) and the y-axis to (-50, 50). This example serves as a demonstration of how the DWA guides the robot efficiently, considering factors such as reaching waypoints and avoiding obstacles.

Moreover in the final part it seems that the computed trajectory starts to be a bit unprecise, and it may be dued to the fact that the loop iterates only about 100 cycles.

In fact, if i try to run it with more loops, also decreasing the variable DT, to increase precision, we obtain the desired path:

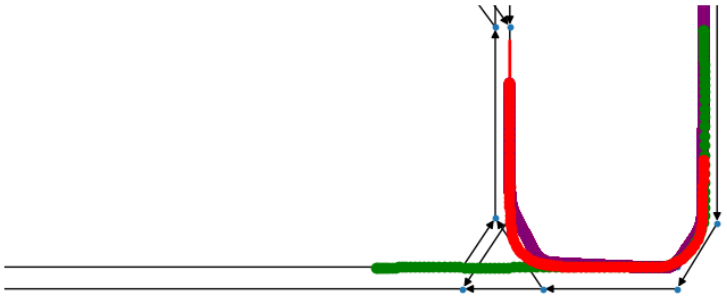

Figure 8: Path given by DWA Algorithm with 1000 iterations

We can also obtain smotheer or sharper trajectories by playing with the weights's costs.

4.2 DWA with Dynamic Obstacle Avoidance

The algorithm presented earlier has been extended to handle dynamic obstacles through the implementation of a Dynamic Window Approach (DWA) with Dynamic Obstacle Avoidance. This enhancement involves the incorporation of a refined cost function that evaluates the trajectory based on distance to the target, speed, and collision penalties, encompassing both static and dynamically moving obstacles. The introduction of a predictive model for dynamic obstacles enables the algorithm to anticipate future obstacle positions, allowing the robot to dynamically adjust its trajectory to evade collisions.

The resulting trajectory is visually demonstrated using the networkx library, showcasing the algorithm's capability to navigate towards a target while intelligently responding to both static and dynamic obstacles.

This adaptation strengthens the motion planning strategy for autonomous robots, particularly in environments characterized by dynamic and unpredictable elements. The final result is shown in the figure below:

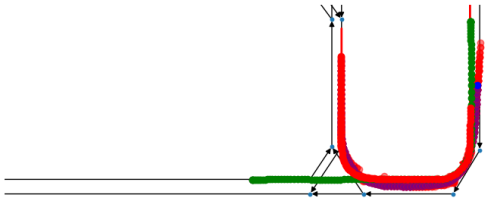

Figure 9: Path given by DWA Algorithm with Dynamic Obstacle detection

In this particular case, if we also look at the printed notes on the code output, we can see that it is never too close to the dynamic obstacle, and the relative cost is always at zero, meanings that it does no affect the motion.

While, in the case we set that we want to maintain a greater distance to the obstacle:

$distance < ROBOT\_RADIUS + OBS\_RADIUS + 50$

Then the final output is the following:

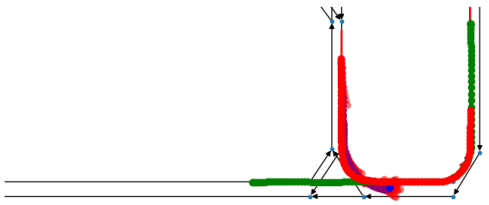

Figure 10: Path given by DWA Algorithm with Dynamic Obstacle detection of increased distance

From which we can observe that, due to the higher distance to maintain from the obstacle, it keeps the velocity of the robot lower, so that in the iteration given by the example (100) it is not able to go further as show in the Figure 9.

5 Conclusion

Embarking on this lab project, we dived into the intricacies of robotic path planning and dynamic obstacle avoidance. Beginning with the fundamental task of loading OpenStreetMap (OSM) maps, we visualized road networks and plotted paths in local coordinates. This laid the foundation for a global planner implemented with a custom A* algorithm, orchestrating optimal routes in a pre-existing road network.

The heart of our exploration was the Dynamic Window Approach (DWA), a local planner enabling a robot to navigate through simulated environments. This journey extended to handling dynamic obstacles, predicting their future positions, and dynamically adjusting the robot's trajectory to avoid collisions. Weighted cost components, including distance, speed, and collision penalties, fine-tuned the robot's response to its surroundings.

Through visualizations, we witnessed the algorithm in action, observing graphs, paths, and trajectories. The incorporation of dynamic obstacles added complexity, revealing the interplay between safety margins and collision detection parameters.

In essence, this project was a journey into understanding the nuances of robotic navigation. We experimented with various scenarios, tested algorithms, and observed how these systems respond to dynamic challenges. This iterative process deepened our comprehension of the complexities inherent in creating effective robotic navigation systems.