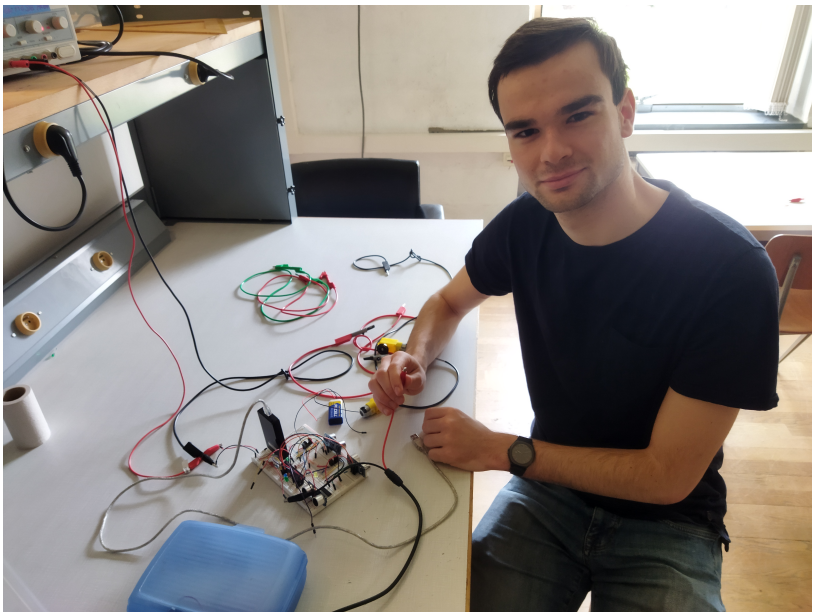
Faculté des sciences appliquées

LIÈGE université
Sciences Appliquées

Introduction to Robotics
Final Project Report
2023-2024



Performed by:

Florent Boxus (s201766)
Maxime Temoschenko (s202613)
Anes Skrijelj (s201769)

Under the supervision of:

Pierre Sacré

Sven Goffin & Jerome Adriaens

1 Introduction

In today's rapidly evolving technological landscape, the integration of autonomous systems into everyday tasks is becoming increasingly prevalent. Autonomous robots have the potential to revolutionize various industries, including logistics, healthcare, and hospitality, by improving efficiency, reducing costs, and enhancing overall service quality. One particularly promising application is in the domain of food delivery.

The global food delivery market has experienced significant growth in recent years, driven by the convenience and demand for quick, reliable service. However, traditional delivery methods face numerous challenges, such as traffic congestion, high labor costs, and inefficiencies in route planning. Autonomous delivery robots offer a compelling solution to these challenges by providing a scalable, cost-effective, and efficient alternative.

This document outlines the objectives and phases of our autonomous robotics project, which focuses on delivering pizzas in an initially unknown environment. The project is divided into three essential phases: (1) mapping the environment and detecting initial ArUco markers, (2) locating the remaining ArUco markers, and (3) delivering the pizza in the most optimal manner.

In the first phase, our autonomous robot is tasked with exploring an unfamiliar environment. Utilizing state-of-the-art Simultaneous Localization and Mapping (SLAM) technology, the robot constructs a comprehensive map of the area. During this exploration, the robot also identifies and records the positions of any ArUco markers it encounters. These markers are critical for subsequent navigation and delivery tasks.

The second phase involves a thorough search for any ArUco markers that were not detected during the initial mapping. The robot employs optimized search algorithms and advanced sensor fusion techniques to ensure that all markers within the environment are accurately located and logged. This step is crucial for enhancing the robot's navigational accuracy and ensuring efficient delivery routes.

In the final phase, the robot leverages the mapped environment and the precise locations of the ArUco markers to deliver pizzas in the most optimal way possible. By employing pathfinding algorithms and optimization techniques, the robot plans and executes delivery routes that minimize travel time and energy consumption. The ultimate goal is to ensure that pizzas are delivered swiftly and accurately, maximizing customer satisfaction.

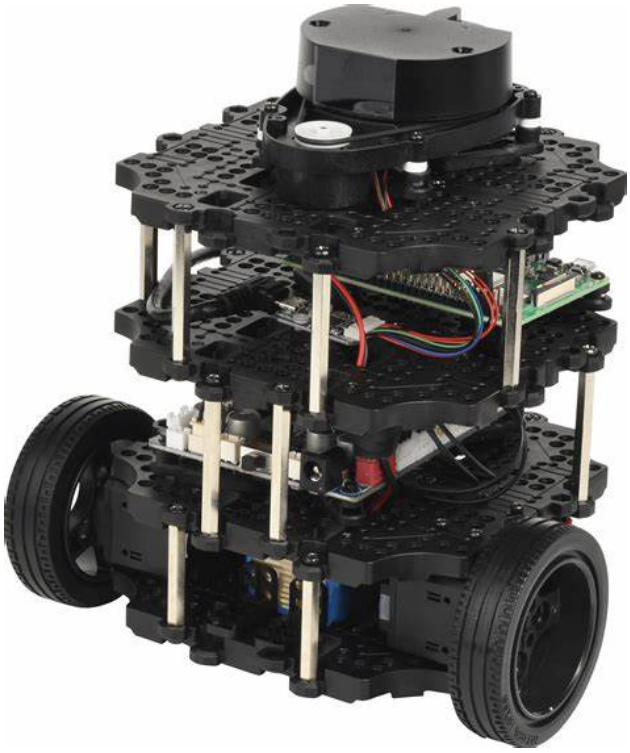


Figure 1: turtlebot we used for this project

2 Mapping

2.1 Nav2 Librairy

The Nav2 library, also known as Navigation2, is a comprehensive and versatile navigation framework for robots running on the Robot Operating System 2 (ROS 2). It is designed to enable autonomous navigation in diverse and dynamic environments, providing a robust solution for robotic applications in various fields, such as logistics, agriculture, healthcare, and domestic robotics. Nav2 offers a suite of tools and algorithms that facilitate path planning, obstacle avoidance, and localization, making it a cornerstone of modern robotic navigation systems. We used the action planner of nav2, providing usefull functions such as feedbach mechanisms and it is able to notify when the goal is reached, justifying why we used action to map the environment.

2.1.1 Key Components of the Nav2 Library

The Nav2 library consists of several key components that work together to achieve efficient and reliable navigation. In order to launch the nav2 node, we needed to lanch à priori the cartographer. Here are the components we used.

2.1.2 Path Planning

Nav2 includes a variety of path planning algorithms that allow a robot to determine the most efficient route from its current location to a desired goal. These algorithms take into account the robot's kinematic constraints, the environment's layout, and any dynamic obstacles that may be present. Popular algorithms such as A*, Dijkstra, and RRT (Rapidly-exploring Random Tree) are supported, providing flexibility in choosing the appropriate method for specific applications.

2.1.3 SLAM

It gave us an implementation of the SLAM using the LIDAR sensors reading to provide a real time mapping. In order to do that efficiently, it provides efficient obstacles avoidance mechanism and position evaluation.

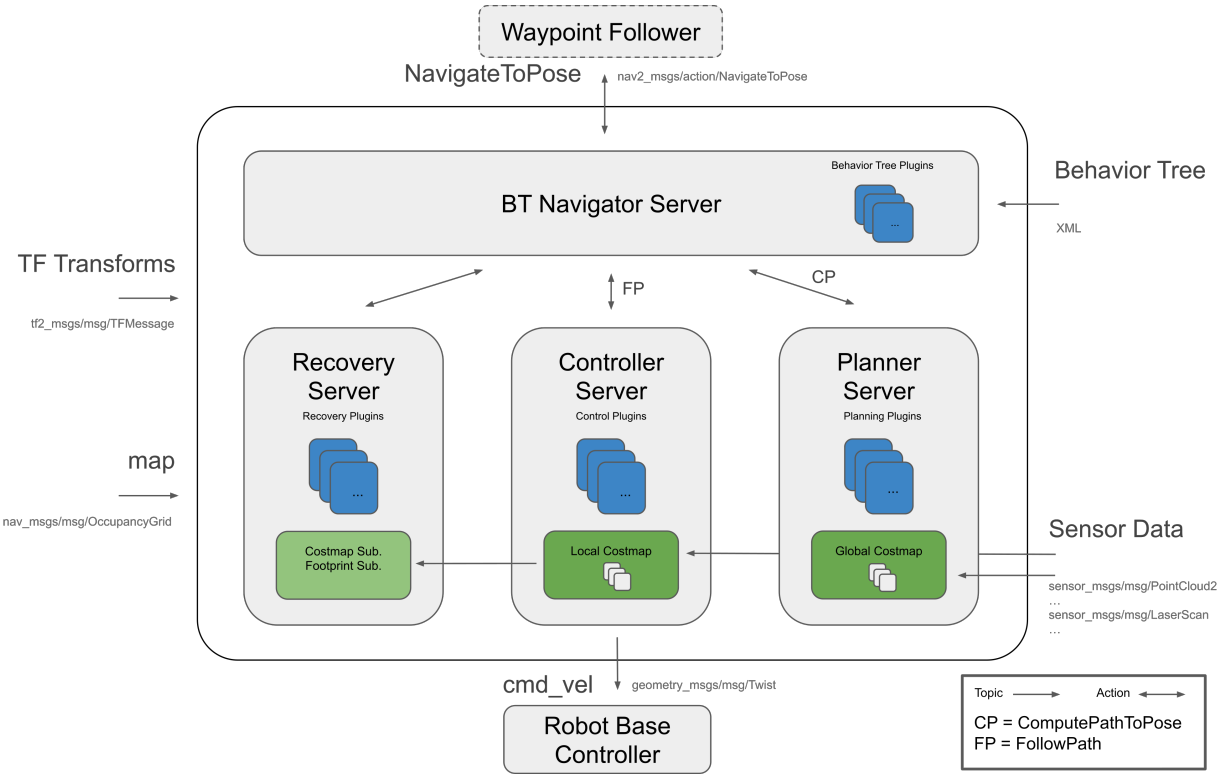


Figure 2: Nav2 Librairy

2.2 Occupancy Grid

The nav2 librairy works with OccupancyGrid. An occupancy grid is essentially a two-dimensional array where each cell corresponds to a specific area in the environment. The value stored in each cell represents the probability or likelihood of that area being occupied by an obstacle. An occupancy grid is characterized by:

- **Resolution:** The size of each cell in the grid, which determines the granularity of the map. Higher resolution grids provide more detail but require more computational resources.
- **Obstacle Values:** Cells can have values ranging from 0 (completely free) to 100 (completely occupied), -1 when unknown.
- **Updating:** As the robot explores the environment, sensor data is used to update the grid, refining the map's accuracy.

We get these informations by subscribing to the "map" topic and reading "OccupancyGrid" messages from it. By processing this message, we get its resolution, size and values in each point representing the percentage of likelyhood there is an obstacle at this point, or -1 if this point is unexplored. Before we think about implementing any algorithms for exploration, we need to first retrieve information about the map the robot is currently seeing. We do this by subscribing to the "map" topic and reading "OccupancyGrid" messages from it. When a new message from said topic is received, we enter the "listener_callback method in order to process the information from said message and store it in the "MyOccupancyGrid" variable that will be used in the rest of the Code.

Do note that the map topic is posting messages almost continuously, which can pose problems to us. If we process the occupancy grid and want to take an action, but midway through the computation the occupancy grid is changed, this could cause problems. To alleviate this issue, we only update the Occupancy grid once the robot has reached its temporary goal. In order to explore the map, throughout its journey, the robot will get a series of temporary goals. It is only once a goal has been reached, that the robot updates its internal occupancy grid and decides on its next action based on it.

2.3 Preprocessing of Occupancy Grid

The Occupancy Grid returns for each spot a value either between 0 and 1 (mapping the probability times 100 to find an obstacle there) either -1 if that spot is unknown.

We decided to consider each point so that:

- If it is more than 50 or has a direct neighbour higher than 50, we set to 1.
- If it is -1 or has a neighbour worth -1, we set it to -1.
- Otherwise we set it to 0

Therefore our grid is made only of 3 possible value and it will be much easier to deal with it.

2.4 Potential Goal Matrix

The robot is very sensitive to walls, when it gets too close, it enters recovery mode and basically does everything possible except the behaviour we expect from it. As a first stage to protect us against that phenomenon, we decreased the inflation radius to 0.2. But this is not enough.

What we have observed is that the robot behaves much better if we send it in areas that have been covered by its lidar and considered as obstacle free zones. Therefore we decided to send it only in the area considered safe. To do that we create the potential goal matrix. To build it, we consider each point of the occupancy grid and we set each any point to 0 if:

- There is a -1 in a range of 4 elements from it or the element is -1
- There is a 1 in a range of 8 elements from it or the element is 1.

The other elements are set to 1 and represents the potential goals toward which the robot can be safely sent.

2.5 Frontiers identification and clustering

Now we want from the preprocessed Occupancy Grid to identify the potential frontiers, ie the area acting as a boundary between explored and unexplored areas.

To do that we create a copy of the processed occupancy grid but we fill it with zeros. Once this is done, we look in the processed occupancy grid for points marked as 0 with a -1 in their direct neighbour(also diagonal ones). If there is one -1, we set to 1 the corresponding spot in our new matrix.

The `label` function from scipy.ndimage is used in image processing and computer vision to identify and label connected components in a binary array. Here's a brief explanation of its purpose and operation:

- **Purpose**: The `label` function scans a binary array (where elements are either 0 or 1) to find and label groups of connected 1s, which are referred to as connected components or clusters.
- **Operation**:
  1. **Input**: A binary array where 1s represent the elements to be clustered.
  2. **Output**: A labeled array of the same size as the input array, where each connected component of 1s is assigned a unique integer label. The function also returns the number of distinct clusters found.
  3. **Algorithm**:
     - The function starts from the first element and scans the array row by row.
     - When it encounters a 1, it initiates a search (using techniques like flood fill or union-find) to mark all 1s connected to it as part of the same cluster.
     - Each newly discovered cluster is assigned a unique label.
     - This process continues until all elements of the array have been examined.

Once we have our clusters (a matrix of the same shape than occupancy grid where each element notes the cluster to which it belongs), we need to identify the center of these.

As a first step, we drop the all the clusters with insufficient number of elements that we consider as noise, we set this threshold to 7 element to consider the cluster as a valid frontier, this was determined empirically.

Then to get the middle of the cluster, we take the medium x position and Y position.

Finally, we want to set as a next goal the middle of the closest cluster from the robot in order to avoid too long and inefficient trips. So we need to convert the spot position in the occupancy grid into the position used by the cartographer. This is done by the following equations:

$$x_{map} = (x_{grid} * resolution) + x_{originMap} \tag{1}$$
$$y_{map} = (y_{grid} * resolution) + y_{originMap} \tag{2}$$

We then perform a loop to identify the closest center of clusters from the robot position.

2.6 Making the link with potential goal matrix

Now, remember that we can't set direclty this position as a next goal, because it's likely to be in an unexplored area, or really close to it. This is the all point of having creating the potential goal matrix as we will now look for the reachable element the closest from the closest cluster center by simply computing distance between each reachable point and each center of cluster. We therefore have our x and y coordinates for the next goal.

2.7 Sending the goal

We now pass the coordinates to be reached to the action server client. Note that we didn't specify yet any orientation to be reached (we will in the final phase of the project). The action allows to have continuous feedback on the robot position and notifies when the goal is reached with some tolerance.

Once it reaches the goal, we update the occupancy grid, find all the new frontiers, determine the next goal and the cycle continues until we have mapped the whole environment.

3 Aruco localization

3.1 Rotating the robot

Many times during the implementation of the project we were tempted to make the robot rotate since the camera as only a 30° view range. We tried first to interrupt goals during execution but it appeared it was a bad idea because not possible to implement efficiently with nav2. Then we switched to cmd vel topic to communicate directly with the motors. We had to reduce the rotating speed in order to avoid drifting on the map. We performed complete rotations of the robot by controlling the time it takes to rotate given its speed. Unfortunately, even if it was able to spot the arucos while rotating, it was not able to transform correctly their position with the mechanism explained bellow. So we finally have 2 solutions:

- We do not do any rotation, we might miss some arucos in the neighbourhood but the application of the algorithm will eventually find them. It can make us win some time since the rotations are quiet slow and therefore time-consuming.
- We make it rotate until it finds an aruco, then it stops, mark a half second pose than go to the next goal. It's a bit slower than the first one but it can be more efficient than the first in terms of distance travelled.

We will alternate between those two solutions.

3.2 Transfrom analyzis

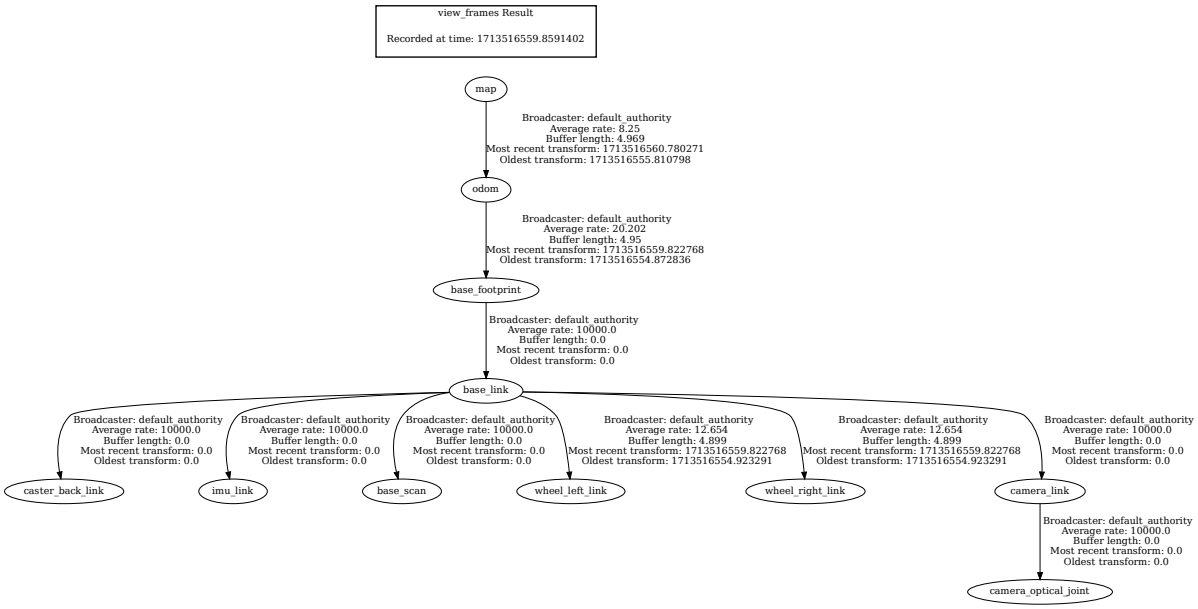


Figure 3: Tf2 tree

Figure 3 shows the tree with all the transformed that must be carried out to get the aruco positions in our map. The aruco package publishes the position of the aruco with respect to the camera optical joint frame. We therefore have a chain of 5 transforms to do in order to get the position in the map frame, that nav2 will be able to exploit.

First, we check if the transform between each of these 6 frames is ready, otherwise we disgard the measurement.

After that, we look up for the transform between the map frame and the camera optical joint and we transform the position given by the aruco package with the transformation matrix. We then extract the position and orientation of the aruco in the map frame. To reach the same configuration than figure 4, the

position to be reached is calculated so:

$$x_{new} = x_{aruco} - 0.5 * cos(\theta + \frac{\pi}{2}) \tag{3}$$

$$y_{new} = y_{aruco} - 0.5 * sin(\theta + frac\pi 2) \tag{4}$$

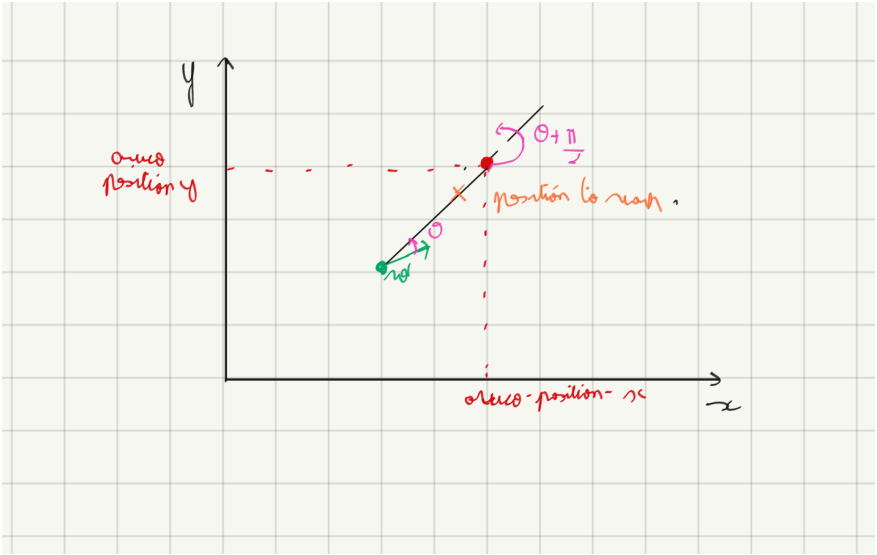


Figure 4: New position

We decided to update the position of the aruco by taking the average of all the calculated positions for the same aruco, providing additionnal robustness with respect to the noisy measurements.

3.3 Algorithm to find arucos

During the mapping phase, we already try to find all the aruco but usually, when the mapping is iver, all of the 10 aruco are not found. Therefore, we added another algotithm which continues until all arucos are spotted and that sometimes allow to finish the mapping. It consists in setting goals the furthest possible from where the rotations (or the simply the ancient goals ,if we decide to drop rotation) have been done. It makes use of the potential grid to set the next goal and calculates the furthest reachable point with respect to all the ancient goal. There fore, it will theoritically keep on indifinitely until it has found all the arucos.

4 Task 2 : Pizza delivery

For the task of delivering the Pizzas, the robot has to find the shortest route visiting each customer once, in order to minimize the delivery time. This problem is known as the **Travelling Salesman Problem**.

4.1 Model of the problem

The problem is represented as graph where the nodes correspond to the robot position at the beginning of the second task and the positions where the robot delivers the pizzas.

Figure 5: Occupancy grid with the robot position in red and positions to deliver in purple

4.1.1 Estimation of the distance between nodes

In order to model the graph, the distances between are computed on the occupancy grid. Attention must be drawn to the fact that this is a two-dimensional map with obstacles that must be taken into account when calculating distances.

The chosen algorithm is $A*$ algorithm, computing distance from a source node to a destination node on a weighted graph because the heuristic allows to consider fewer nodes and thus being more computing efficient than non-guided path finding algorithms.

The graph where distance calculations are performed is a graph where each cell of the computation grid is a node and the weight between adjacent node is 1 or $\sqrt{2}$.
The strength of this algorithm is the use of an heuristic to guide the path finding. In this context, the heuristic is the euclidean distance between the node of the grid considered and the goal.

In a few words, at every iteration, one considers the node with the lowest estimation distance from the source node to the destination node, going through the considered node (at initialization, this is the source node). The estimation is the sum between the best known distance from the path to this node and the heuristic from the considered node to the goal. From the considered node, one updates the neighbouring nodes estimation cost and add it to the possible future candidate nodes if the estimation found is better than previous one. The algorithm stops when the considered node is the destination node.

Figure 6: A* Algorithm between 2 nodes

4.1.2 Salesman Problem

The distances between the nodes are known. One searches to minimize the route visiting all nodes once. The optimal algorithm is an algorithm testing each permutation of the nodes and selects the permutation with the smallest distance. The main drawback of this approach is the complexity of the algorithm $O(n!)$ where n are the number of nodes. As the number of customers vary, the computation time fluctuates, resulting in a non-scalable algorithm despite finding the optimal shortest path.

```
Algorithm BruteForceSalesman(dists)
  Input: A dictionary dists where keys are pairs of nodes (i,j) and values are the distances
  d_ij between them.

  Output: The minimum distance and the optimal path

  nodes ← list of all nodes
  optimal_path ← null
  distMin ← +inf

  for each permutation perm of nodes do
    if the first node of perm is not robot then
      continue
    end if
    permDist, permPath ← result of permComputation(perm, dists)
    if permDist < distMin then
      distMin ← permDist
      optimal_path ← permPath
    end if
  end for
  return distMin, optimal_path
end Algorithm
```

Therefore, the nearest neighbor algorithm is considered. In this heuristic, the next chosen node is the node whose distance is the shortest from the considered node and not yet visited. The main advantage is the complexity $O(n^2)$, resulting in a more scalable than the naive solution. However, there is no guarantee that the final path is the optimal path (in average it is 15% longer than the optimal path).

```
Algorithm NNSalesman(dists)
  Input: A dictionary dists where keys are pairs of nodes and values are the distances between them
  Output: The minimum total distance and the path
```

```
  nodes ← list of all nodes
  sizeNodes ← number of nodes
  visited ← empty set
  current_node ← 0
  path ← empty list
  total_distance ← 0

  while the number of visited nodes is less than sizeNodes - 1 do
    nearest_distance ← +inf
    nearest_node ← null

    for each neighbor in nodes do
      if neighbor is not the current_node and neighbor is not in visited then
        distance ← dists[(current_node, neighbor)]
        if distance < nearest_distance then
          nearest_distance ← distance
          nearest_node ← neighbor
        end if
      end if
    end for

    append (current_node, nearest_node) to path
    total_distance ← total_distance + nearest_distance
    add current_node to visited
    current_node ← nearest_node
  end while

  return total_distance, path
end Algorithm
```

Nodes	0	1	2	3	4	5	6	7	8	9	10
$n!$	1	1	2	6	24	120	720	5040	40320	362880	3628800
n^2	0	1	4	9	16	25	36	49	64	81	100

Table 1: Evolution of the associated complexity of Bruteforce and Nearest neighbor Heuristic as the number of node increases. The bruteforce approach does not scale.

After analysis, the nearest neighbor approximate solution seems more appropriate to implement because computation times are reduced and the average path found is not much longer than the optimal path.

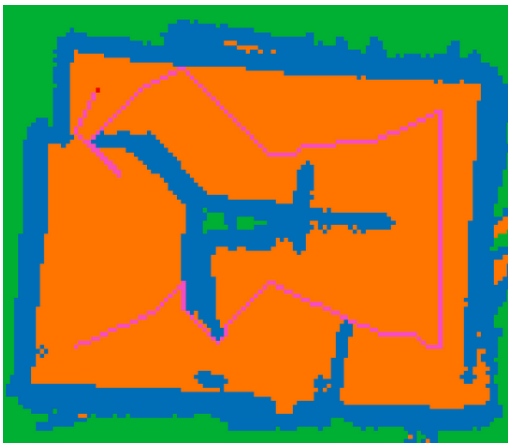


Figure 7: Bruteforce algorithm : Distance : 307.29, Computation time : 90s

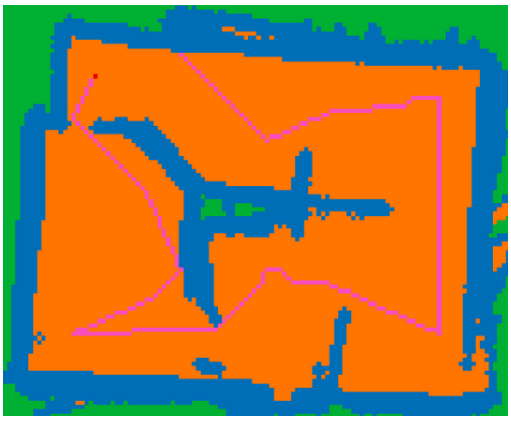


Figure 8: Nearest Neighbor Heuristic : Distance : 314, Computation time : 2s

Figure 9: Comparison of bruteforce and NN on seven delivery positions

5 Possible Improvements

There can be several possible improvements that can be conducted :

- Playing with nav2 parameters to reduce the risk of crashes.
- Improving the communication between the command vel topic and nav2 action. It would have allowed us to make complete turns for spotting several arucos at once.

6 Conclusion

In this project, we developed and implemented functions for preprocessing occupancy grids, identifying frontier cells, and planning paths based on the detection of ArUco markers. The main objectives and outcomes of our project can be summarized as follows:

- **Preprocessing Occupancy Grids**: We created a preprocessing function to refine occupancy grid data by categorizing cells based on their proximity to obstacles or unexplored areas. This step ensures that our grid data is clean and ready for further analysis.
- **Frontier Detection**: We designed the `searchFrontiers` function to detect and label frontier cells. These are critical areas that lie adjacent to unexplored regions or obstacles, which are essential for navigation and exploration tasks in robotics.
- **Labeling Function**: We utilized the `label` function to efficiently identify and label connected components in the binary array produced by our frontier detection. This allowed us to group frontier cells into distinct clusters, facilitating the analysis of frontier regions.
- **Finding ArUco Markers**: Our system includes functionality to detect all ArUco markers within the environment. These markers serve as key points for navigation and localization, providing reference points for the robot.
- **Path Evaluation and Planning**: After identifying all ArUco markers, the system evaluates potential paths and determines the best possible route for the robot to take. This involves considering the positions of the ArUco markers, the identified frontier cells, and the current state of the occupancy grid to optimize the navigation path.

The implementation of these functions demonstrates our ability to process and analyze occupancy grids, detect important navigation markers, and plan efficient paths for robotic movement. By identifying frontier regions and leveraging ArUco markers, our system can assist robots in planning exploration paths, enhancing their ability to navigate unknown environments efficiently.

Overall, this project lays the groundwork for advanced exploration and navigation strategies in robotics, highlighting the importance of preprocessing, frontier detection, marker-based localization, and path planning in autonomous systems.