SOAR PROJECT GROUP A

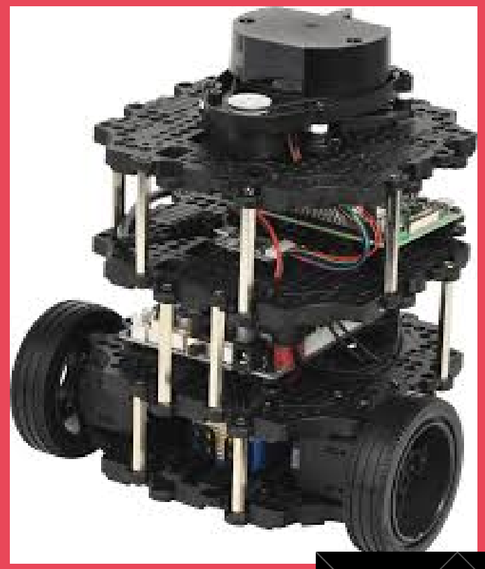
SIMONE LUIS MUHUHO
BARNABAS MATRAI

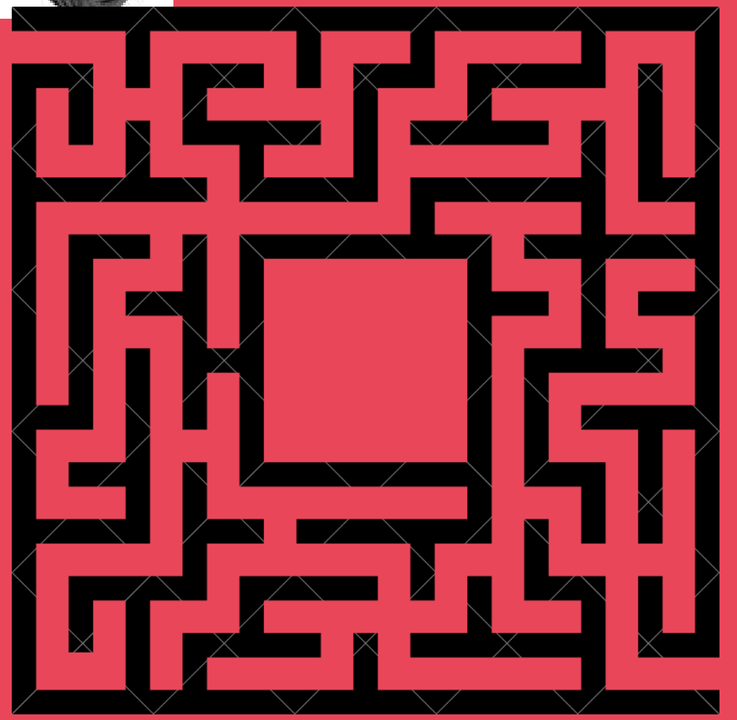
SCUOLA MEDIA ORIANI

LEZIONE DI SCIENZE

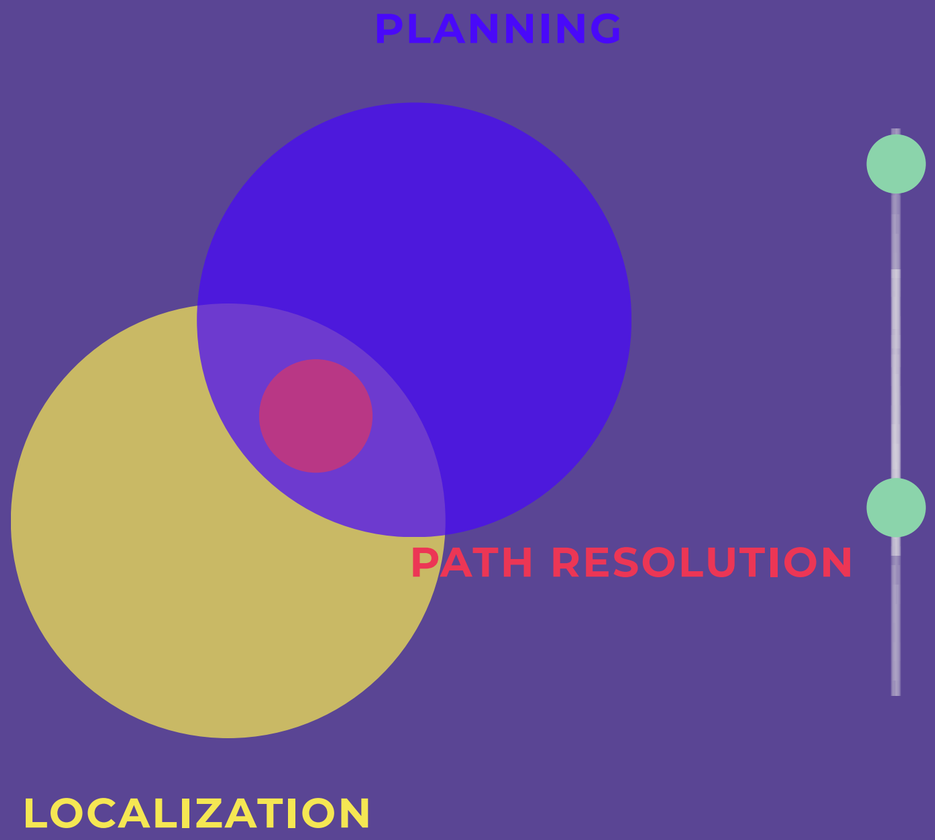
WHAT IS THE GOAL?

- THE OBJECTIVE FOR THIS PROJECT WAS TO CREATE AND CODE 2 NODES IN EITHR C++ OR PYTHON FOR THE ROS1 THAR WOULD ALLOW THE TURTLEBOT3 TO LOCALCAZIE AS WELL AS NAVIGATE ITSELF OUT OF A MAZE, DYNAMICALY CALCULATING THE PATH OUT OF SAID MAZE.





THE PROCESS



01 LOCALIZATION

The first step is to get a set of informations from the robot that are needed to even undestand where the robot is. this majorly includes things like a data structure that can then be reshaped to visualize the map as well a the scann data. these are then gonna get reused by tools such as the KNN aproximation model to give a better estimate of the starting position, which is then gonna get sent to the other node.

02 PLANNING

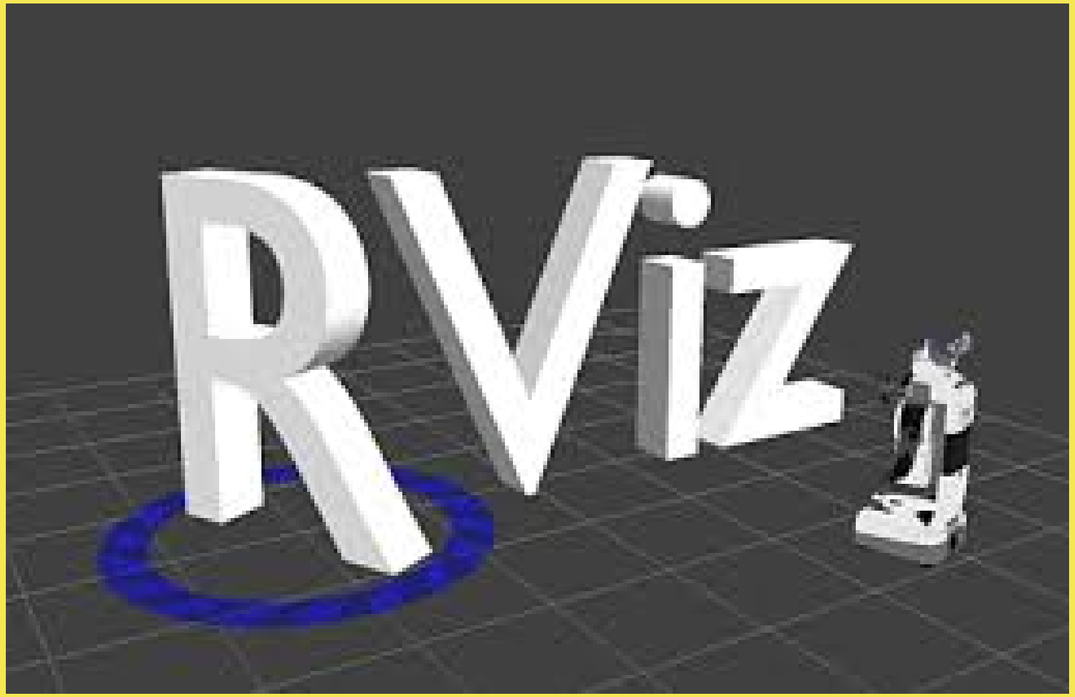
After reciving the robots initial position and keeping it monitored for every change the node creates the edges in between the maps nodes to form the full map. Next, the a* algorithm processes the map based on the robots postion as well as the goals postion finding the best path in according to its criteria. Finally the goal and path are posted onto ROS and the robot starts moving.

THE PROCESS

HTTPS://GITHUB.COM/SIMONEMUHUHO/ROS1-SOAR/BLOB/MAIN/ROS%20PROJECT.DRAWIO.PNG

BUT HOW DO WE USE OUR NODES

AS REQUESTED BY PROJECT REQUIREMENTS WE USED A LAUNCH FILE FOR OUR PROJECT. THIS NOT ONLY STRAIGHTFOWARDS THING BUT IT ALSO ALLOWED US TO CONSIDER SOMEINTRESING OPTIONS




```
<node pkg="ROS1-SOAR" type="localization.py" name="localization" output="screen" />
<node pkg="ROS1-SOAR" type="planning.py" name="planning" output="screen" >
    <param name="goal" value="$(arg goal)" />
```

LOCALISATION

- LOAD SIMULATED LASERSCAN AND MAZE MAP
- TRANSFORM LASERSCAN FROM POLAR COORDINATES TO IMAGE
- MATCH LASERSCAN IMAGE TO MAP
- GET ROBOT’S STARTING POSITION

PREPARING THE KNN DATA

```python
def prepare_knn_data(map_array, rec_map):
    h, w = map_array.shape
    X, y, free_cells = [], [], []

    for i in range(h):
        for j in range(w):
            wx, wy = map_to_world(j, i, rec_map)
            label = 1 if map_array[i, j] > 50 else 0

            X.append([wx, wy])
            y.append(label)

            if label == 0:
                free_cells.append((wx, wy))

    return np.array(X), np.array(y), free_cells
```

- Converts the occupancy grid map into training data for kNN
- Labels each map cell as free space or obstacle
- Stores free-space locations as valid robot positions

LOCALIZING THE ROBOT

```python
def localize_robot(scan_points, knn, free_cells):
    best_score = -1
    best_pose = None

    scan_points = push_away(scan_points)

    for rx, ry in free_cells:
        transformed = scan_points + np.array([rx, ry])
        predictions = knn.predict(transformed)
        score = np.sum(predictions == 1)

        if score > best_score:
            best_score = score
            best_pose = (rx, ry)

    return best_pose
```

- Tests all possible robot positions in free space
- Shifts the laser scan to each candidate position
- Uses the kNN map model to classify scan points
- Selects the position with the best match to walls

PLANNING

- READS THE ROBOT'S STARTING POSITION
- DOES A SEARCH FOR A PATH TO THE GOAL LOCATION
- PUBLISHES THE PATH TO /MOVE_BASE, NAVIGATING THE ROBOT TO THE GOAL

CREATING THE GRAPH

```python
def has_horizontal_edge(self, node1, node2):
    # Check for obstacles between two nodes on the same row
    assert node1.my == node2.my, "Nodes must be in same row"
    y = node1.my
    for x in range(min(node1.mx, node2.mx) + 1, max(node1.mx, node2.mx)):
        if self.grid[y, x] == 1:
            return False
    return True

def has_vertical_edge(self, node1, node2):
    # Check for obstacles between two nodes on the same column
    assert node1.mx == node2.mx, "Nodes must be in same column"
    x = node1.mx
    for y in range(min(node1.my, node2.my) + 1, max(node1.my, node2.my)):
        if self.grid[y, x] == 1:
            return False
    return True
```

- Checks if two nodes can be connected
- Scans the map cells between the nodes
- Rejects the edge if any obstacle is found

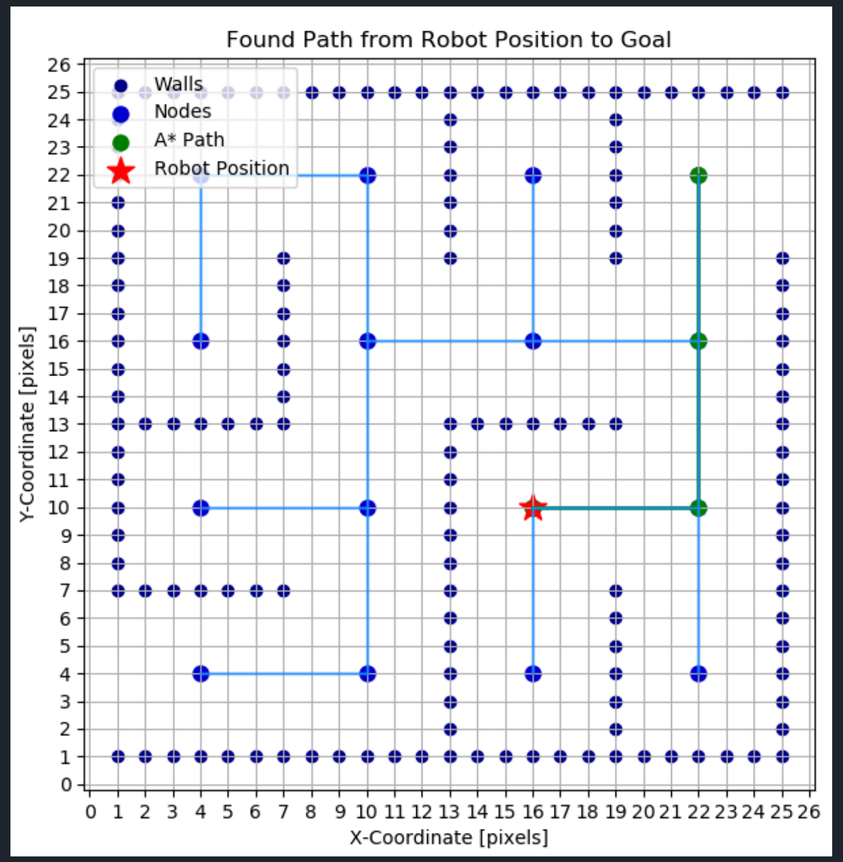
A* ALGORITHM

```python
def a_star(self, start_node, goal_node):
    # Perform A* pathfinding
    open_set = set()
    closed_set = set()
    parent = {}

    open_set.add(start_node)
    goal = goal_node
    found = False

    while open_set:
        current = self.sort_h(open_set)
        if current == goal:
            found = True
            break

        open_set.remove(current)
        closed_set.add(current)

        for neighbor in current.neighbors:
            if neighbor in closed_set:
                continue
            if neighbor not in open_set:
                open_set.add(neighbor)
                parent[neighbor] = current
```

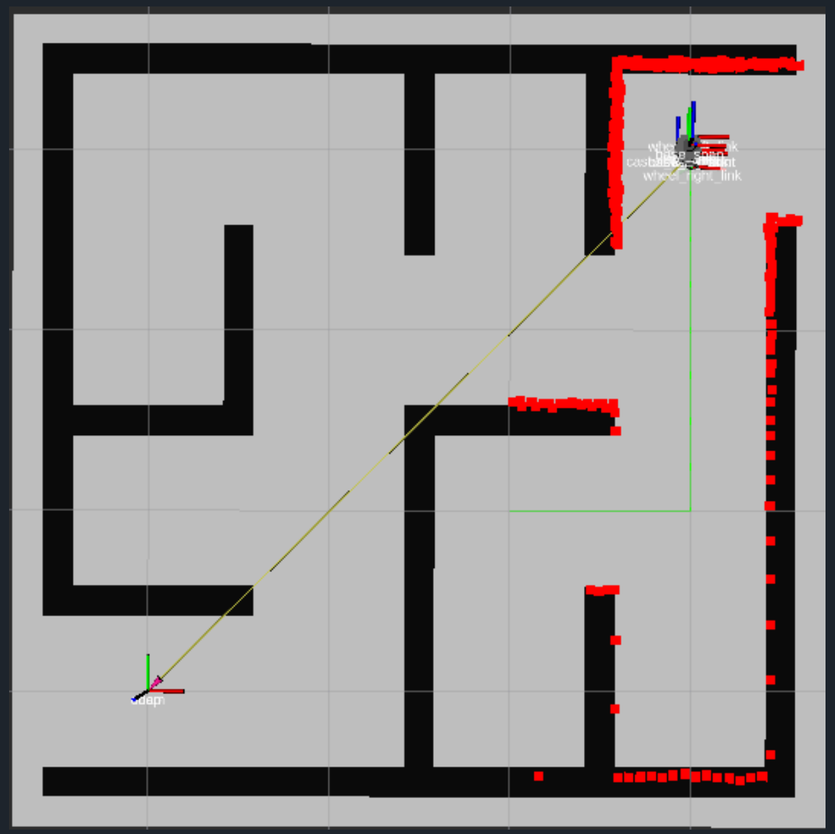
```python
# Path reconstruction
if found:
    path = []
    node = goal
    while node != start_node:
        path.append(node)
        node = parent[node]
    path.append(start_node)
    path.reverse()

    # Publish path and poses
    self.publish_path(path)
    for node in path:
        bx, by = self.map_to_block(node.mx, node.my)
        self.pose_pub.publish(self.make_pose(bx, by))

    return path
else:
    return None
```

- Performs A* search to find a path from start to goal
- Uses Manhattan distance as the heuristic
- Expands nodes with the lowest estimated cost first
- Reconstructs the path once the goal is reached
- Publishes the final path and goal poses

RESULTS



Visualization of the path found



Robot after navigating to goal in RViz

FUN FACTS

- DFS (DEPTH FIRST SEARCH) functionality is still in our code but innactive
- KNN finishes before the we subscribe to the laserscan
- Maptest.py was a file used to debug map functionalities


Thank you!