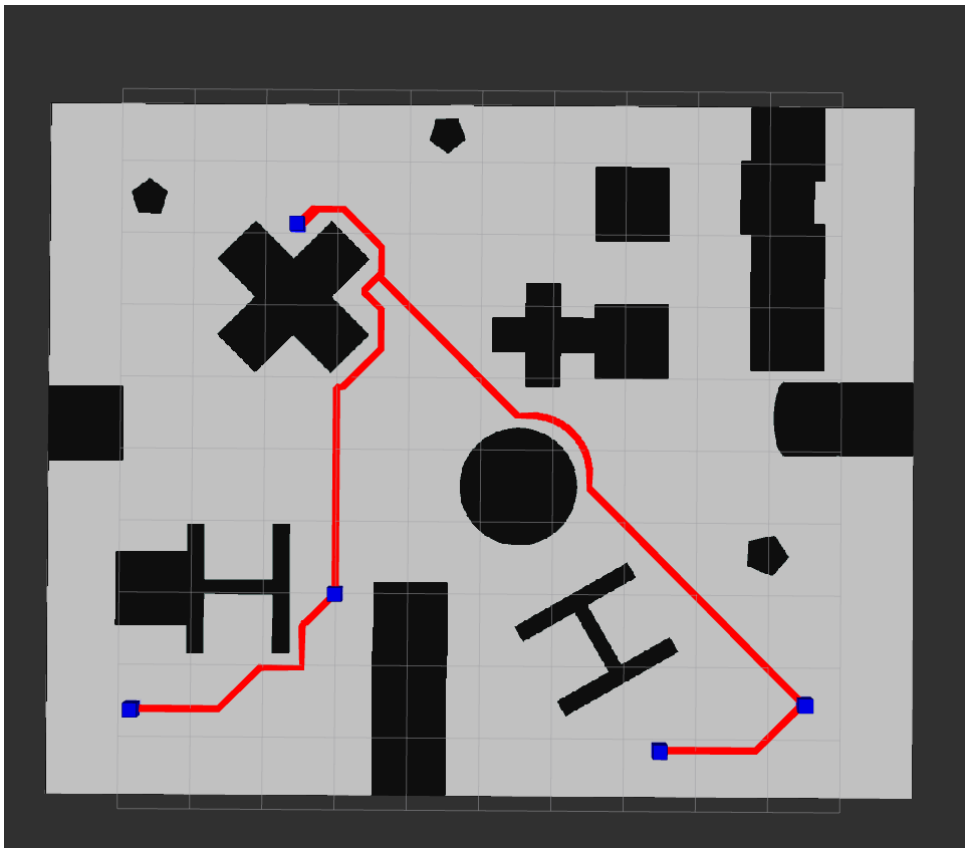

# [CS3027]

## [Robotics Assessment]

[Bradley Scott] | [51661169 | [b.scott.16@aberdeen.ac.uk]

# Task 1 – Using RViz, nodes and transforms

Dummy localization was amended to broadcast a transform of /base_pose_ground_truth and /map frame called /real_robot_pose.

I have also displayed the path the robot must follow and goals(including start) in rviz as markers.

I have also displayed the robot's true position and orientation using base_pose_ground_truth, with an rviz marker.

# Task 2 – Path Planning

- Map interpretation (Appendix 1.1)

This task involves several components working together to create a path suitable for visualizing and driving in rviz.

Firstly, the map data must be interpreted for my path planning algorithm to use. To do this I pulled the map occupancy grid from the map topic which ros publishes using the map files provided for the assessment.

I then converted the map from its 1d representation into a 2d representation, using the map width and height information provided by the map topic, so that It could be used more easily for the rest of the assessment.

Finally, I had to inflate my grid, to make sure that the robot did not strike any obstacles. This is because I am performing my path planning on a pixel representation of the map. Therefore, If a path is drawn within 8 pixels of an obstacle, the robot will crash into that obstacle as the robot is around 8 times larger than a pixel.

The perfect inflation would be 8 pixels in every axis surrounding each occupied edge pixel of an object.

My inflation however is much larger, at 15 pixels. This is due to odometry errors (see section 3 - Driving).

Goal 4(the gap goal) however, requires an inflation rate of no more than 5 pixels, anything larger will close the gap. For this reason, I was not able to drive to this goal. However, I was still able to plan a path to the path (appendix 1.2).

To debug my inflation, I created a script that converts the grid to a png representation (see appendix 1.3). This allowed me to work out the maximum amount of inflation that could occur before the gap for goal 4 was closed.

- Creating a Path (appendix 2.1)

I have successfully managed to create the shortest path between the start and goal points, by solving both the travelling salesman problem and implementing a path planning algorithm.

My travelling salesman solution uses the path planning algorithm A* to work out the distance between all pairs of nodes so I will discuss A* first.

- Implementation of A*(appendix 2.2)

A* uses a heuristic to prioritize the expansion of nodes in the right direction.

My neighboring function only checks for neighbors on the horizontal and vertical axes of the node, not diagonal neighbors. This was a design decision to improve the runtime of my A* Algorithm.

The next step was to work out the order in which to apply my A* implementation to visit the goals. This problem is a form of the travelling salesman problem. Since there are only 5 nodes in my instance (start, goalo..goal3) there are only 120(5!) permutations of goal orderings. This is not very computationally expensive in this instance so I can afford to take a brute force approach to the problem.

Performing A* on 120 paths on the other hand would be quite computationally expensive and time consuming so I simplified the problem further.

Rather than performing A* 120 times, I only have to perform it for all combinations of pairs in the set. Meaning I only have to perform A* a total of 10 times.

Once I have my combinations and permutations of the set of goals. I can even further simplify the problem. Rather than checking the total cost of visiting 120 paths I can half this number by removing any reverse permutations from my list of paths. This is because the cost of getting from say 1,2,3,4 is the same as getting from 4,3,2,1 So I don't have to calculate this twice. As a result, I only have to calculate the cost of 60 paths using my node combinations.

Finally, I set the best path to the minimum costing path that starts at the starting node (best path returned doesn't always start at the start node).

- Path Smoothing (appendix 2.3)

My path so far contains several hundred nodes which will make driving very difficult and inaccurate due to errors in odometry.

To overcome this problem, I created a path smoothing algorithm which removes nodes in the path by only keeping nodes which represent a change in direction. Taking my final path down to less than 80 nodes. Making driving more accurate and manageable.

## Task 3 – Driving (appendix 3.1)

To gain attain accuracy and reduce error in my driving I went with the following approach:

Calculate angle to goal, calculate distance to goal, rotate to face that angle, move that distance, stop, repeat

To reduce errors in odometry and retain accuracy in reaching goals I found that the slower the robot moves, the less error there will be. This is because of the natural deceleration of the robot when a stop command is received. Meaning, a higher speed results in a higher stopping distance, which makes the robot overshoot the goal. Overshooting the goal with minimal inflation means that the robot will crash into obstacles.

Given more time I would have improved my driving to make it smoother (without starting and stopping) by implementing a proportional controller. A proportional controller allows the robot to slow down as it approaches the target, allowing the robot to move at a faster speed and calculate the rotation on the move.

## Task 4 — Localisation

For localisation I am using the ros package amcl. Amcl creates a map obtained from laser information and outputs pose estimates by creating a particle filter.

Fake_Localization is an api used by amcl and must be ran in parallel. I have displayed where amcl thinks my robot is in rviz as a marker.

## Conclusion

Given more time to improve my implementation further I would refactor my code so that my modules are independent ros nodes. Meaning I could launch all my modules, path planning, driving etc from the launch file. My nodes would have to be structured in such a way that they would wait for start messages before executing. Allowing them to run in the required sequence. I would also pass all path data as messages, rather than txt files as this is not very good programming practice.

Another addition would be to remove goals on rviz once they have been reached by the robot.

Please see (appendix 5.1) for running instructions of my project.

Appendix

| Index | Description | Location |
|---|---|---|
| 1.1 | Map interpret and inflation script | /src/robotics/scripts/maplisten.py |
| 1.2 | 5 goal planned screenshot | /GAP.png |
| 1.3 | Grid to png script | /src/robotics/scripts /gridmaker.py |
| 2.1 | Path planning implementation | /src/robotics/scripts /pathplanning.py |
| 2.2 | A* implementation | /src/robotics/scripts /astar.py |
| 2.3 | Path smoothing implementation | /src/robotics/scripts /pathsmooth.py |
| 3.1 | Driving implementation | /src/robotics/scripts /drive.py |
| 5.1 | Running instructions | /readme.txt |